Data Recovery of Distributed Hash Table with Distributed-to-Distributed Data Copy*

Yusuke DOI††,††, Member, Shirou WAKAYAMA†, Nonmember, and Satoshi OZAKI†, Member

SUMMARY To realize huge-scale information services, many Distributed Hash Table (DHT) based systems have been proposed. For example, there are some proposals to manage item-level product traceability information with DHTs. In such an application, each entry of a huge number of item-level IDs need to be available on a DHT. To ensure data availability, the soft-state approach has been employed in previous works. However, this does not scale well against the number of entries on a DHT. As we expect $10^{10}$ products in the traceability case, the soft-state approach is unacceptable. In this paper, we propose Distributed-to-Distributed Data Copy (D3C). With D3C, users can reconstruct the data as they detect data loss, or even migrate to another DHT system. We show why it scales well against the number of entries on a DHT. We have confirmed our approach with a prototype. Evaluation shows our approach fits well on a DHT with a low rate of failure and a huge number of data entries.

key words: Distributed Hash Table, data durability, traceability system, scalability

1. Introduction

In Distributed Hash Tables (DHTs), such as Chord [1] and Pastry [2], data stored in the system are often managed in soft-state styles [3]. In other words, data entries on a DHT are stored only for a certain period of time, and data sources with the primary data must refresh them before the corresponding entries on the DHT expire.

Soft-state mechanisms are simple to implement and manage. However, a problem arises if the number of data entries becomes huge. A soft-state mechanism requires refresh messages proportional to the number of data on the system. Typically, a system employing a DHT is a large-scale system, and the amount of data on the system can also be huge.

In this research, we propose a framework to enable Distributed-to-Distributed Data Copy (D3C). With the framework, a system can recover from data losses due to node failures, cooperating with original data sources. For example, we have been working on an ID index system for item-level product traceability systems [4]. The objective of this research is to make the index system practical by realizing automatic data recovery from node failures. In addition, we evaluate D3C against a model of product traceability systems to confirm it scales well against the number of data entries.

1.1 Related Works

A typical distributed product traceability system is based on the model of EPCglobal. In the EPCglobal standards, ONS (Object Naming Service) [5] is used to bind RFID tags with corresponding databases. Because ONS is based on the Domain Name System, it is difficult to handle item-level information on ONS and compliance with the ONS standard means the system does not resolve item-level information. The objective of our previous work and Huang’s work [6] was to replace ONS with a DHT system in order to make the system robust and sufficiently scalable to handle item-level information appropriately.

Researches [7]–[9] have been done to make DHT more robust and available by using replications in a DHT. These works are on stand-alone reliable peer-to-peer storage. For example, Carbonite [10] focuses on durability of data on DHT. With delayed recovery against transient failures of replicas and reintegration of failed replicas that returned to the DHT, Carbonite manages the number of replicas efficiently within the range that can keep sufficient availability.

Although the replication mechanisms can reduce the risk of data loss, they are imperfect. Data will be lost if the situation is worse than the assumption of the replication system. Hence, even the replication approach works well, users of a DHT system need a way to recover from data loss.

Unlike storage applications, data on the DHT system are just an index of related information in the product traceability system. Participants in the traceability system have original data and they can regenerate the index. Our approach enables detection of lost data and recovery. Our aim is to recover the lost data from another data source. Hence, a simple, fragile, cheap, and scalable DHT system can be used as the index.

We previously presented the basic design of D3C [11]. In this research, we evaluate the design according to a design of distributed traceability system [12].

1.2 Scenario: DHT for Item-Level Product Traceability

We used a DHT system to index a huge number (up to $10^{10}$ order) of RFID tags for each product item. Figure 1 shows the
structure of the proposed system in a beef-product traceability case. Concrete data of each item, such as a production date, IDs of raw materials, and records of temperatures in warehouses are managed in databases owned by each participant (local DBs in the figure). The ID-DB resolution service resolves ID to DB location and a DHT system is in the service. Keys used in the DHT are generated from IDs given by the RFID tag and the corresponding value is the list of URLs of local DBs with concrete data of the tagged product. Note that nodes in DHT are in a managed environment (e.g., data centers) and the risk of node failure is a controllable parameter. Similar arguments are also discussed in [13] and they conclude that using peer-to-peer style client-side machines to make an available DHT system is not practical.

2. D3C: Distributed-to-Distributed Data Copy

In this section, we describe Distributed-to-Distributed Data Copy (D3C) approach to recover data entries against DHT node failures.

Figure 2 shows the idea of D3C. A DHT-based copy target $\beta$ is a distributed index. Distributed data source $\alpha$ are a set of independent data sources with primary data to be indexed. The structure of D3C makes the recovery/copy process distributed and scalable against the number of data in $\beta$.

In this section, first we define the objective of D3C. Then, we describe structure and use case of D3C for traceability case in Sect. 2.2. The system consists of two parts. Distributed data source $\alpha$ is described in Sect. 2.3 and DHT $\beta$ as the distributed copy target is described in Sect. 2.4.

2.1 The Goal of D3C

From our target scenario described in Sect. 1.2, we define the objective of D3C as follows. The objective is to overcome the scalability problem of soft-state based approaches and the imperfection problem of replication approaches.

The Objective:
To realize data durability on DHT with sufficient scalability against the number of data entries.

At the same time, we assume DHT nodes are well managed and the failure rate is as low as that of a typical server. In addition, we accept temporary loss of data entries on DHT. Our approach does not realize continuous data availability against data loss. Our approach is to let primary data sources of the data entries on DHT recover the required data efficiently in order to revive the entries. Relation with data mirroring for continuous data availability is discussed in Sect. 5.1.

2.2 The Case for Traceability System: Recovery from Data Loss

For the traceability system described in Sect. 1.2, we adopt D3C to recover data losses with DHT node failures. Figure 3 describes recovery of lost data by detecting node failures of DHT $\beta$. The data loss and recovery process starts node failure incident in the DHT $\beta$ at the top. The lost hash-space region is advertised to each of distributed data source $\alpha$. Using Copy Trigger, Filter, Sorter, and Bulk Uploader, each data entry in the region is re-uploaded in certain time period.

2.3 Distributed Data Source $\alpha$

Distributed data source $\alpha$ is a group of independent data sources that has some data index to be copied to a DHT system $\beta$. Typically, each independent data source corresponds to a step in product traceability chain in Fig. 1.

As shown in Fig. 3, each independent data source in $\alpha$ has the following components. Some of them are optional.
in some use cases. The use case of data recovery for traceability systems is described in Sect. 2.2.

- **Copy Trigger**: It detects the necessity of a copy and initiates a copy process.
- **Filter**: Based on Copy Trigger’s decision, it selects data entries to be copied to β.
- **Sorter**: It sorts output of the Filter in hash key space of copy target β to optimize the data transfer.
- **Bulk Uploader**: It uploads the data from Sorter to corresponding node in β.

In the following, we describe functions and effects of the modules.

The Copy Trigger initiates a copy action in α. It must detect the algorithm of β such as hash function and hash key adjacency in hash key space, an endpoint information (IP address and port number) of a node in β to start the copy, and a filter condition such as hash key region or update time that defines the data copied to β. The way to detect them and the condition to initiate a copy are defined based on the application’s request of D3C. The use case in traceability application is described in Sect. 2.2.

The Filter selects the entries needed for this copy action using a condition found by the Copy Trigger. If the Copy Trigger gives a region as the filter condition, it filters out data entries outside the region from the data source by calculating h(key) using h() as the hash function of the β. If the clock is given as the filter condition, it checks the last update time of each entry and picks entries fresher than the given clock. The filtered output will be passed to the Sorter as tuples of (h(key), value).

The Sorter sorts the input stream of the updated dataset: the unsorted list of (h(key), value). The sort order is based on the hash value adjacency of the hash space of the β. The sort order can be obtained from the Sorter. In the case of Chord and Pastry, the space is linear and simple ascending sort will work. It may also change or randomize the start value (i.e. zero point) of the hash value of the sorted list to avoid concentration of data transfer requests from many data sources in α on the same β node at the same time.

The Bulk Uploader uploads the list of data entries sorted by the Sorter effectively. It can access the copy target DHT node lookups are not required.

Figure 4 is the simplified Bulk Uploader algorithm in Java. In the code, the sorted_source is an iterator of (hashed key, value) tuple, and a DHTNode instance represents a remote DHT node. A DHTNode instance raises RedirectException if it receives requests on a hash value outside the node’s authoritative region.

For the first run of the algorithm, nptr points an arbitrary node of β and a series of put operation and RedirectException works as a recursive lookup of the corresponding node of the first entry in the sorted_source. After the recursive lookup, a series of put operations will be performed successfully without redirection.

2.4 Copy Target DHT β

In D3C, a copy target DHT β is the system that receives the copy of data from α. In this paper, we focus on Chord-like algorithms. However, D3C uses adjacency of hash value space only and many similar DHT algorithms such as Pastry are acceptable.

The β is just a DHT system. However, use cases may require extensions to invoke Copy Triggers of α.

For example, we have implemented Shared Table in our prototype to maintain and share information about lost data regions. In the data recovery case, new lost data region invokes Copy Triggers. This is required for the loss-recovery use case described in Sect. 2.2. Next, in this section, we describe why and how to maintain the Shared Table in DHT β.

The information needed for the Copy Trigger should be advertised to independent data sources of α by some mean. If the sources need to contact one or a small number of servers to know the information, the servers will become a single point of failure or a bottleneck. To avoid that, D3C utilizes copy target DHT β to advertise the information. We implemented the Shared Table, an append-only data table shared among DHT nodes with orthodox peer-to-peer flooding. The flooding algorithm makes all entries written to the Shared Table appear in all DHT nodes. In this section, we describe how to use the Shared Table to circulate informa-
tion needed by the loss-recovery case described in Sect. 2.2.

Lost region entries are written to the Shared Table. A lost region entry is described in \((\text{start:end}]\) where \text{start} and \text{end} are hash values to describe the lost region. If a DHT node detects a lost region, it appends a lost region entry corresponding to the region. If the node has a mirror of the region, it may append the last update time of the mirror. In that case the entry will become \((\text{start:end], lastUpdateTime})\).

Every entry written in the Shared Table must have a time stamp, an entry ID including the node ID that created the entry, and a time-to-live value. Because each entry has an ID and a time stamp, lost region entries on the same region at different times will be distinguished.

In the Shared Table, every row has a well-known key such as \text{NodeInfo} (to inform new node arrival), or \text{LostRegion} (to inform lost regions). Entries are appended to the Shared Table of a node, and appended entries are copied to other nodes recursively. Each node keeps the entries until they expire.

All entries appended on the Shared Table of a node will be distributed to neighbor nodes. In Chord, the successor node, nodes in the successor list, and targets in the finger table are the neighbor nodes. When a node receives a record update from other nodes, the node records the entry to its own Shared Table. The node recursively sends the newly arrived entry to its neighbor nodes only if the entry is new. Hence, a node does not send duplicated entries and entry broadcast loop is avoided.

With the Shared Table described in this section, lost region entries of copy target DHT \(\beta\) will arrive at all DHT nodes of \(\beta\). Each independent data source in \(\alpha\) can find updated lost region entries and invoke the \text{Copy Trigger} by polling a node in \(\beta\) in an appropriate interval (i.e. less than TTL of the lost region entries).

2.5 Lost Region Detection in Chord

In many DHT algorithms, including Chord, the data region that should be managed by a node is decided by the relation with neighbor nodes. Hence, the region would be increased or decreased by the change of neighbor node. Upon decrease, the decreased region can be taken over by another node and no problem arises. On the other hand, the node must take over the expanded region if a neighbor moved away.

Figure 5 depicts the case in a Chord-like algorithm. In Chord, a region is managed in a linear hash value space, and the region managed by a node is decided by the node and its predecessor node. In Fig. (a), there are three nodes: \(N_1\), \(N_2\), and \(N_3\). Each manages data regions: \(R_1\), \(R_2\), and \(R_3\), respectively. If \(N_2\) drops, the region \(R_2\) will be lost (marked ‘\(X\)’ in Fig. (b)). A Chord node regularly tries to contact its successor node to find if the successor is alive or not. As \(N_3\) finds \(N_2\) is gone, it switches its successor pointer to the next successor \(N_1\), and notifies its presence to \(N_1\). Then, \(N_1\) notices \(N_2\) has gone and \(N_3\) is the predecessor for now.

After that, \(N_1\) changes its predecessor pointer from \(N_2\) to \(N_3\). \(N_1\) notices it must manage \(R_2\) and the region has been lost (Fig. (c)).

3. Analysis

In this section, we analyze D3C and compare scalability between it and soft-state based approach. As our target is traceability systems, we need to understand scalability against the data scale.

3.1 D3C and Soft-State Based System

On soft-state systems, the order of traffic amount is controlled by the number of data entries \(D\) and the number of DHT nodes \(N\). A DHT request is sent for each entry refresh. Each request involves \(O(\log N)\) messages between DHT nodes in Chord-like algorithms. As the refresh is needed on a per-entry basis, the total amount of traffic is \(O(D \log N)\).

On the other hand, we need to consider the following to estimate the order of the traffic amount for D3C-based loss recovery: the Shared Table traffic to distribute the lost region entries, polling of the Shared Table from each independent data source of \(\alpha\), and the traffic to recover the lost regions.

First, we estimate the order of the traffic on the Shared Table on \(\beta\). A lost region entry is sent to every neighbor node of each node on \(\beta\). In Chord, the order of the number of neighbor nodes is \(O(\log N)\). In the Shared Table, each node in \(\beta\) sends the entry once to all the neighbors. Hence, an entry in the Shared Table is distributed by \(O(N \log N)\) messages in \(\beta\).

Let \(R\) be the possibility of a node failure in a unit time. The average number of node failure in a unit time in the \(\beta\) DHT system is \(N \cdot R\). Hence, the order of the amount of messages to update the Shared Table entries of lost regions is \(O(R \cdot N^2 \log N)\).

The traffic amount on \(\beta\) is modeled as follows. It consists of the monitoring traffic of the Shared Table and the recovering traffic against the lost regions. As we assume the

\[\text{Traffic} = O(D \log N) + O(N \log N) + O(R \cdot N^2 \log N)\]
number of independent data source in $\alpha$ is proportional to the amount of data $D$, the amount of monitoring traffic is also proportional to $D$.

The amount of recovering traffic for a lost region is $O(\frac{D}{R})$. This is the average amount of the lost data for each failure. By applying the average number of failed nodes in the DHT system $\beta$, the order of recovering traffic becomes $O(D \cdot R)$.

Hence, comparing soft-state systems and our D3C-based loss recovery, soft-state systems scale well against the number of nodes $N$. For $N$, soft-state-based systems scale in $O(\log N)$ and D3C-based systems scale in $O(N^2 \log N)$ for $\beta$ and $O(1)$ for $\alpha$. For $D$, both approaches are in the same order: soft-state systems are in $O(D)$ and D3C-based systems are in $O(1)$ for $\beta$ and $O(D)$ for $\alpha$.

However, D3C is better if $R$ can be controlled and the node failure risk is small as $R$ is applied to the traffic amount except the monitoring traffic of the Shared Table. On the contrary, on systems with high risk of node failures such as a DHT consisting of client PCs, D3C will cause disastrous traffic for the Shared Table updates with the order of $O(N^2 \log N)$ against the number of nodes.

### 3.2 Estimations on Large Scale Traceability Scenario

As described in Sect. 1.2, the target application of this research is large-scale traceability system. Here, we evaluate scalability in the following two models: one-segment model and challenging model.

The one-segment model is based on the number of convenience stores in Japan. We assume the number of shops is 40,000 and each shop has 4,000 kinds of items, according to some information on the web\(^1\).

The challenging model is a huge system with $10^{10}$ IDs. The order is derived from e-Japan plan\(^2\) of Japanese Cabinet. In a previous work\(^3\), we estimated 10,000 DHT nodes are enough to serve $10^{10}$ order DHT-based naming system for traceability systems.

Combining the two models, we defined two sets of $D$, $N$, and $C$: the number of local traceability DBs (Fig. 1) in the system. The parameters are shown in Table 1. Note that $D$ of the one-segment model is product of the number of shops and the number of items per shop. $N$ is guessed based in the same way with the previous work. On the contrary, $C$ of the challenging model is estimated with an assumption that $C$ is proportional to $D$.

The failure rate, $R$, is guessed from a survey\(^4\). In this estimation, we use 4 failures per year as the average number of server failures. In addition, we allow 72-hour of data unavailability at most. In other words, refreshing in soft-state system should take at least once per 72 hours.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>One-Segment Model</th>
<th>Challenging Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>160,000,000</td>
<td>$10^{10}$</td>
</tr>
<tr>
<td>$N$</td>
<td>200</td>
<td>10,000</td>
</tr>
<tr>
<td>$C$</td>
<td>40,000</td>
<td>2,000,000</td>
</tr>
</tbody>
</table>

To compare D3C against soft-state model, we guess the number of logical sessions in the system. Upon the analysis defined in Sect. 3.1, we guess the approximate number of sessions in the traceability system as Table 2. Note that the number of sessions is for the whole traceability system. Due to the large amount of individual data, it is clear that the soft-state model is difficult to employ in this scale of traceability system.

### 4. The Prototype and Experiments

In this section, we evaluate D3C in terms of traffic amount with our prototype.

The evaluation environment consists of 8 PCs: NEC Express5800 110Ba-m3 (CPU: Intel Pentium M 1.8 GHz, Memory: 512 MB). and the PC for the control is a DELL PowerEdge 1850 (CPU: Intel Xeon 3.2 GHz, Memory: 512 MB). A dedicated 1000 Base-T network segment connects them and no major traffic other than the evaluation exists on the segment.

#### 4.1 The Prototype

We have implemented a prototype that corresponds to the loss-recovery case. The DHT part for $\beta$ is a home-brewed implementation based on the Chord algorithm. The distributed data source for $\alpha$ is a simple prototype to register a URL on every predefined product ID as an index and keep the records against node failures.

On the DHT side, we developed two functions. One is the Shared Table described in Sect. 2.4. In addition to the table management module and entry update message, we developed a Shared Table lookup protocol to let independent data sources in $\alpha$ poll the Shared Table in $\beta$.

The other is a function for the evaluation that imitates data losses. As described in Sect. 2.5, a node detects a lost region if the predecessor has gone. In addition, to imitate the data loss, our prototype accepts table refresh requests. Upon such a request, a node will erase the contents of the internal table for DHT (not the Shared Table), and writes a new data loss entry of itself.

The prototype of data source $\alpha$ is just a proof of concept. It is called dummyDB. It imitates a local DB in the distributed product traceability system in Fig. 1. A dummyDB has its service URL and a list of IDs. At the startup, it registers the URL to keys corresponding to each product ID in DHT $\beta$. After that, a poller thread starts to monitor if there are new lost region entries. Once a new lost region

---

\(^1\)http://en.wikipedia.org/wiki/Convenience_store.


\(^3\)DOI et al.: DATA RECOVERY OF DISTRIBUTED HASH TABLE WITH DISTRIBUTED-TO-DISTRIBUTED DATA COPY

is detected, it invokes the **Filter** to select the IDs in the region, and the **Bulk Uploader** to recover the data. After the recovery, it returns to the steady polling state.

With the combination of our modified DHT implementation and the dummyDB, we confirmed the loss-recovery use case.

### 4.2 The Effect of **Sorter**

We evaluated the effectiveness of bulk transfer with the **Sorter**. We measured the amount of traffic for bulk uploading of a set of data entries, and compared the amount from random order data source against one from sorted data source by the **Sorter**.

Against 81 DHT nodes (10 for each PC plus 1 for bootstrapping) running on the 8 PCs, 2000 random data are put from another PC with random and sorted order. Just before the upload begins and just after it ends, we measured the number of packets sent/received on network interfaces and the elapsed time. Table 3 shows the result for an average of 3 trials for sorted order and random order, respectively. We confirm that the **Sorter** reduces both the number of packets and the time required for bulk transfers.

In this evaluation, measured elapsed time is not a realistic value. We assume the value is reliable enough for the comparison between soft-state and D3C with the following reasons.

The time may affected by OS context switching and loopback network communication. As the uploading program is a single-threaded program on an independent PC and only $N$ or $N + 1$ times of recursions occurs in the sorted upload case, we consider validity of the elapsed time in random upload case.

Context switching slows down the random upload if recursive lookup goes to the same node in about 1/10 chance. Loopback communication results faster recursion on the same node. As each effect interfere the other, we assume the impact is limited.

### 4.3 Prototype D3C and Soft-State System

As our objective is to solve the scalability problem of soft-state approaches against data scale, we evaluate the data-loss recovery case of D3C described in Sect. 2.2 against soft-state approaches. Note that D3C is fundamentally different from soft-state approaches and direct comparison is difficult. In this research, we evaluate scalability against the number of data entries on systems through the order of traffic amount required.

As scalability evaluation, we have conducted a series of lab-scale experiments.

The experiments are for cases of 10,000 to 100,000 data entries against 54 to 108 DHT nodes. We observed the number of packets sent and received in the same way as described in Sect. 4.2.

For soft-state cases, all the entries are added in an interval to emulate the refresh effect of soft-state mechanisms. For the D3C cases, 4 instances of dummyDB manage a quarter of the data entries and try to update the lost regions.

Node failures are emulated by re-initializing tables in nodes of $\beta$. A table of a random DHT node is initialized with average interval $I$. With varying $I$, we observe the effect of node failure. The duration of each experiment is 600 seconds. In the followings, D3C-I, D3C-II, and D3C-III corresponds to $R = \frac{1}{81 \times 15}$, $R = \frac{1}{81 \times 30}$, and $R = \frac{1}{81 \times 60}$, respectively.

Figures 6 and 7 show the number of messages exchanged in the system per second per data entry. They describe the efficiency against $D$. In these figure, horizontal line such as soft-state case (SS) shows linear scalability against $D (O(D))$. Other lines are D3C. They show the scal-

![Fig. 6](image)

**Fig. 6** Recovery efficiency against data size: $N = 54$.

![Fig. 7](image)

**Fig. 7** Recovery efficiency against data size: $N = 108$.

†In average, one node be re-initialized per every 15 seconds in $N = 81$ experiment. For different $N$, re-initialize interval is set to be same $R = \frac{1}{\frac{N}{15}} = 0.000823$ (failure/sec).
ability against $D$ because it keeps the data on DHT more efficiently for larger $D$.

5. Discussions

In this section, we offer on some considerations on D3C.

5.1 The Relation between Mirroring and D3C

Data mirroring among DHT nodes is the best known way to hide DHT node failures. However, mirroring does not ensure the data availability against simultaneous DHT node failures. Hence, mirroring and D3C are not an exclusive way to make DHT practical. D3C without mirroring increases the number of visible node failures and increases $R$ significantly. With combination of mirroring and D3C, a system will be highly available against isolated failures and durable against simultaneous failures.

5.2 Security Considerations

An attacker may try denial-of-service by inserting faked lost region information. To avoid that, lost region information reported by non-adjacent nodes against the region should be ignored. In addition, each DHT node may check connectivity and uptime of nodes around the adjacent node. A DHT system with public key-based node ID may enforce signature on each item of lost region information.

6. Conclusion

In this research, we have evaluated D3C (Distributed-to-Distributed Data Copy) to recover lost data due to DHT node failure efficiently. We focused on loss-recovery use case of D3C with an ID-DB resolution system for a traceability system in mind. Although On the other hand, D3C can be used to initiate a new DHT system from distributed data source, or to make transition between two different DHT systems.

From analysis and evaluations, we conclude that D3C fits better if the number of data is large and DHT nodes are reliable. There’s a tradeoff between D3C and soft-state approach. Tradeoff analysis is given in Sect. 3.

For the traceability case, we made two models: the Challenging model and the One-Segment model. Based on the models, we have estimated traffic scale. In addition, we have measured real traffic amount of lab-scale prototype to understand scalability against the number of data entries. Through the evaluation, we confirmed our approach fits better than soft-state approach. This is because we assume we can control node reliability to suppress worse scalability against the number of node failures. As the future works, we improve scalability against the number of node failures. Definitely, we apply better broadcasting scheme for Shared Table distribution, or we may consider centralized implementation such as dedicated publish-subscribe broker to notify failures to the distributed data sources.

References


Yusuke Doi

received a MS in Media and Governance from Keio University, Japan, in 2000. He is currently a Research Scientist at Corporate Research and Development Center, Toshiba Corporation. At the same time, he is a graduate student of the University of Tokyo. His research interests include distributed systems, naming architectures, and privacy on RFID systems.
Shirou Wakayama received a MS in Media and Governance from Keio University, Japan, in 2003. He is currently a researcher at Corporate Research and Development Center, Toshiba Corporation. His current research interests include networked home appliances, Internet services, and ubiquitous networking.

Satoshi Ozaki received a PhD in electrical and computer engineering from Yokohama National University, Japan, in 1995. He is currently a Research Scientist at Corporate Research and Development Center, Toshiba Corporation. His research interests include ubiquitous networking, Internet services and network security of embedded systems.