IEICE TRANS. INF. & SYST., VOL.E93–D, NO.6 JUNE 2010

PAPER

# Incremental Digital Content Object Delivering in Distributed Systems

**Lung-Pin CHEN**[†a)], *Nonmember*, **I-Chen WU**[††], *Member*, **William CHU**[†], **Jhen-You HONG**[†], *and* **Meng-Yuan HO**[†], *Nonmembers*

**SUMMARY** Deploying and managing content objects efficiently is critical for building a scalable and transparent content delivery system. This paper investigates the advanced incremental deploying problem of which the objects are delivered in a successive manner. Recently, the researchers show that the minimum-cost content deployment can be obtained by reducing the problem to the well-known network flow problem. In this paper, the maximum flow algorithm for a single graph is extended to the incremental growing graph. Based on this extension, an efficient incremental content deployment algorithm is developed in this work.

***key words:*** *content delivery network, incremental algorithm, maximum flow*

## 1. Introduction

In a content delivery system, a service provider is a host responsible for distributing content objects to the client machines. Intrinsically, an object can be delivered via a direct network transmission. However, with such a naive approach, an object has to be entirely retransmitted even if only a small change has been made.

In many Internet applications, it is useful to enhance the performance of content delivery by involving client-side computations. For example, *transcoding* operation [1], [2] is a client-side computation to construct an object based on its predecessors. The total cost of deploying a set of content objects can be reduced by taking the trade-off between direct transmission and client-side transcoding. In [1], the content delivering problem is modeled as a combinatorial optimization problem which can be transformed to the well-known network flow problem [3]–[8]. Based on this trams formation, the optimal content deployment strategy can be efficiently obtained by using the existing network flow algorithms [1], [9], [10].

Due to the diversity of Internet applications, it would be desirable to provide an efficient way of *incrementally* deploying the content objects between clients and servers over networks. For example, in an e-learning system, usually a user accesses the course content in an incremental manner, according to the learning progress, instead of accessing the entire course content at one shot. In this paper, we study the *incremental content object deployment problem*. We first introduce the notion of *incremental flow networks*, which is a sequence of incrementally growing flow networks $(N_1, N_2, \cdots, N_k)$. Then, we show that an incremental content object deployment problem can be efficiently solved by using our new incremental maximum flow algorithm.

The rest of this paper is organized as follows. In Sect. 2, we define the basic and incremental content delivery problems. Section 3 discusses the maximum flow algorithms. Section 4 discusses our incremental content deployment algorithms. Finally, experimental results and conclusions are discussed in Sect. 5 and 6.

## 2. Problem Definition

### 2.1 Basic Content Deployment Problem

A *service provider* (also referred to as a *server*) is a network host that maintains the content database and handles the requests issued by the *clients* (*i.e.* users) over Internet. We assume that the content database is built by a digital content manager and is given ahead of processing time. Also, it is assumed that some *directory service* is provided to facilitate clients to explore and contact the service provider.

The content database is modeled as an *ODG* (*object dependency graph*) $G = (U, F, I, J, net, comp)$, where each $a \in U$ refers to a *content object* (or, simply, *object*) and each link $(a, b) \in F$ refers to the dependency relation from object $a$ to $b$ [1]. We assume that the dependency relation induced by $F$ is *acyclic*. In the ODG $G$, $I$ is the set of *initial objects* which have no predecessor in the graph. Also, $J$ is the set of objects which are designated as the *target objects* to be delivered to the client host. Note that $I \subset U$ and $J \subset U$.

For each object $a$ in $G$, there are two costs associated with it: the cost of transmitting object $a$ from the server to the client, denoted by $net(a)$, and the cost of transcoding object $a$ from all of its predecessors, denoted by $comp(a)$.

Content Deployment

Upon receiving a request for accessing target objects $J$, the server calculates a strategy called *content deployment* for deploying the target objects. Let $U_N$ and $U_C$ refer to the sets of objects that are deployed via network transmission and

Manuscript received October 22, 2009.
Manuscript revised February 25, 2010.

[†]The authors are with the Department of Computer Science and Information Engineering, Tunghai University, Tai-Chung, Taiwan.
[††]The author is with the Department of Computer Science and Information Engineering, National Chiao-Tung University, Hsin-Chu, Taiwan.
a) E-mail: lbchen.con@gmail.com

DOI: 10.1587/transinf.E93.D.1512

client-side computation, respectively. Also, let $U_O$ refer to the objects that are not deployed. A content deployment is a partition $(U_O, U_N, U_C)$ on ODG $G = (U, F, I, J, net, comp)$ that satisfies the following properties:

- All the target objects must be deployed, *i.e.* $J \subseteq \{U_N \cup U_C\}$.
- If an object is deployed via a client-side computation, then all of its predecessors must be also deployed to the client. That is, for all $b \in U$ and $(a, b) \in F$, $b \in U_C \Rightarrow a \in \{U_N \cup U_C\}$.

The *cost* of content deployment $\mathcal{D} = (U_O, U_N, U_C)$ is defined as $cost(\mathcal{D}) = \sum_{v \in U_N} net(v) + \sum_{v \in U_C} comp(v)$. The *minimum content deployment* of $G$, referred to by $MinD(G)$, is the one with the minimum cost among all the feasible content deployments.

The notion of ODG provides a generic framework for interpreting various Internet applications. Consider the example demonstrated in Fig. 1 regarding to an e-book of programming language. The server owns the original copies of content objects of the e-book, all are stored in XML format. According to the ODG problem model, a target object, say $X$, can be delivered from the server to a client via directly transmission. The cost net($X$) refers to the transmission cost of object $X$. According to the problem model, the target $X$ can also be delivered via client-side computation. If this strategy is adopted, the server delivers the predecessor documents of $X$ (in XML format) and the XML transformation script of $X$. Then, the client computes document $X$ by executing the XML transformation script. The cost comp($X$) refers to the time required to perform the XML transformation script for generating object $X$.

In the example demonstrated in Fig. 1, the e-book contains three computer language topics, *data type*, *flow control*, and *function*, each with three types of objects, CONTENT, DISPLAY, and QUIZ. A CONTENT object represents the knowledge of a topic which essentially can not be derived from other objects. Thus, their *comp* costs are set to $\infty$, as shown in the figure. Moreover, their *net* costs should be set proportional to the object size as well as the network bandwidth to reflect the transmission costs.

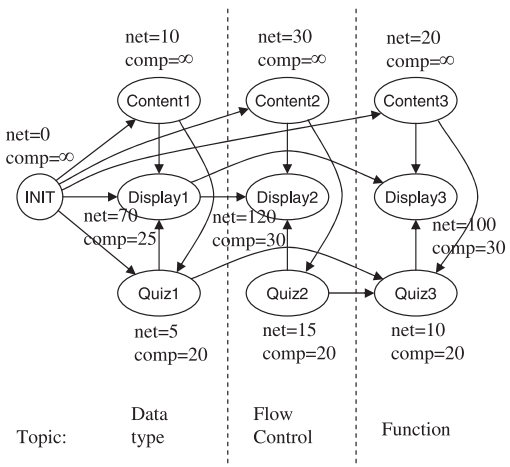



**Fig. 1** An object dependency graph of an e-book of programming language.

In Fig. 1, a QUIZ object represents the learning assessment document, and a DISPLAY object represents the CONTENT object in HTML-format to be rendered in client-side Web browsers. In ODG, a link from node $X$ to $Y$ means that a portion of $Y$ can be derived from $X$. For example, the link from DISPLAY1 to DISPLAY3 in Fig. 1 indicates that some parts in lesson *data type* can be reused in lesson *function* (*e.g.* issues about *parameter passing*). Consider the scenario that a user is intended to access DISPLAY3 object (topic *function* in the e-book) via his/her Web browser. According to the structure of ODG, DISPLAY3 objects can be directly delivered or can be computed from the predecessor objects, CONTENT3, QUIZ3, and DISPLAY1 objects. In the first strategy is adopted, DISPLAY3 object in HTML format is delivered. On the other hand, if the second strategy is adopted, the XML transformation script for DISPLAY3 object as well as the predecessor's documents must be made available to the client.

### 2.2 Incremental Content Deployment Problem

In many Internet applications, one may query the content object database in an incremental order, instead of access randomly. For example, in the above-mentioned e-book application, the user usually accesses the topics in the order of *data type*, *flow control*, *function*. A sequence of requests with such properties is called *incremental* and is formally defined in this subsection.

For an ODG $G$, if there is a path from object $a$ to object $b$ in $G$, denote it by $a \prec b$. (Note that $a \prec b$ implies that $b \not\prec a$ since the graph $G$ is acyclic.) Let $A \subseteq U$ be a set of objects. The set of all objects with paths *to* (or *from*) at least one member in $A$ are called the *upstream objects* of $A$ and referred to by $\mathcal{P}(A)$ (or the *downstream objects* of $A$, referred to by $\mathcal{S}(A)$). For two object sets $A$ and $B$, $A \prec B$ if and only if every object in $A$ is the upstream of some object in $B$, that is, $\mathcal{P}(A) \subset \mathcal{P}(B)$ and $\mathcal{S}(B) \subset \mathcal{S}(A)$. An illustration is shown Fig. 2.

A sequence of ODGs $(G_1, G_2, \cdots, G_k)$, $G_i = (U, F, I, J_i, net, comp)$, is *incremental* if these graphs are all the same except that the target sets are evolved in an successive manner. Formally, $(G_1, G_2, \cdots, G_k)$ is incremental if $I \prec J_1 \prec J_2 \prec \cdots \prec J_k$. Furthermore, we assume that the target object sets are *strictly* incremental, that is, *direct* edge between $\mathcal{P}(J_i)$ and $\mathcal{S}(J_i)$ is *not* allowed. The *minimum*

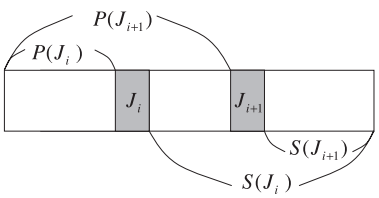



**Fig. 2** Illustration of target object sets $J_i$ and $J_{i+1}$ with strictly incremental property $J_i \prec J_{i+1}$.

*incremental content deployment problem* is to find the minimum content deployment for each member graph $G_i$ in an incremental ODG $(G_1, G_2, \cdots, G_k)$.

## 3. Incremental Maximum Flow Algorithms

This section briefly reviews the maximum flow algorithm [4] and explains its relationship to the incremental maximum flow algorithm. Readers may reference [4] for the minor details of the preflow algorithms.

### 3.1 Flow Network

A *flow network* $N = (V, E, S, T, cap)$ is a five tuple in which $(V, E)$ is a directed graph where each edge $(u, v) \in E$ is associated with the non-negative *capacity* $cap(u, v)$. In the flow network, $S \subset V$ are designated as *source vertices* and $T \subset V$ are designated as *sink vertices*. A source (or sink) vertex only has outgoing (or incoming) edges.

A flow $f$ of $N$ is a function from edges to non-negative values which represents the units of flow been sent from the sources to the sinks without exceeding the edge capacities. Formally, $f$ is a flow on $N$ if and only if the following properties are satisfied[†]:

1. (Capacity constraint) $f(u, v) \leq cap(u, v)$, and
2. (Flow conservation) $IN(u) = OUT(u)$ for each vertex $u \in V \setminus \{S, T\}$,

where $IN(u) = \sum_{v,(v,u)\in E} f(v, u)$ and $OUT(u) = \sum_{v,(u,v)\in E} f(u, v)$.

A *cut* of flow network $N$ is a partition $(X, Y)$ on the vertices of $N$ such that all the source nodes are in $X$-part and all the sink nodes are in $Y$-part. The cost of cut $C = (X, Y)$ is defined as $cost(C) = \sum_{\forall(u,v),u\in X,v\in Y} cap(u, v)$. Note that the only the edges from $X$ to $Y$ contribute their capacities to the cut cost (while those reversed edges from $Y$ to $X$ are *excluded*). The *minimum cut* of $N$, referred to by $MinCut(N)$, is the one with the minimum cost among all cuts of $N$. Ford and Fulkerson prove the following *max-flow-min-cut theorem* [3].

**Theorem 3.1:** Given a flow network, its maximum flow value and minimum cut cost are equal.

■

### 3.2 Generic Preflow Algorithm

A *preflow* $f$ on a flow network is a flow except that the total flow into a non-source non-sink vertex $u$ can exceed that out of $u$. Namely, for all $u \in V \setminus \{S \cup T\}$, $IN(u) \geq OUT(u)$. The difference between the total flow into and out of $u$ is called the *excess flow* and denoted by $ex(u) = IN(u) - OUT(u)$. A *non-source non-sink* vertex $u$ is called *active* if $ex(u) > 0$. Also, an edge $(u, v)$ with $f(u, v) < cap(u, v)$ is called a *residual edge*. Clearly, there is still at most $cap(u, v) - f(u, v)$ units of flow can be sent via edge $(u, v)$ without violating the capacity constraint. For the preflow $f$ defined on a network $N = (V, E, S, T, cap)$, the graph induced by the residual edges is called a *residual network* and is defined as $R(N, f) = (V, E_f, S, T, r_f)$ where $r_f(u, v) = cap(u, v) - f(u, v)$ and $E_f = \{(u, v) \mid r_f(u, v) > 0, (u, v) \in E\}$.

A path in the residual graph is called a *residual path*. Ford and Fulkerson [3] proved Theorem 3.2.

**Theorem 3.2** ([3]): A preflow $f$ is a maximum flow of network $N$ if and only if there is no residual path from sources to sinks on $N$.

In order to maintain the preflow efficiently, a *labeling function* $d$ is used to estimate how close the vertices are to the sinks. For a network $N = (V, E, S, T, cap)$, a *valid* labeling function $d$ is a $V \to \mathbb{Z}$ function, such that $d(v) \leq d(w) + 1$ for each residual edge $(v, w)$. Also, $d(s) = n$ for each source $s \in S$, $d(t) = 0$ for each sink $t \in T$.

Hereinafter, let $n$ and $m$ denote the number of vertices and edges of $N$, respectively. The *generic preflow algorithm* [4] is a class of maximum flow algorithms that maintain a preflow and work by repeatedly choosing active vertices and sending excess flows along residual paths towards sinks. Until no residual path can be found in the flow network, based on Theorem 3.2, the preflow becomes a maximum flow. The generic preflow algorithm is described in Algorithm 1.

#### Multiple-Sink Flow Network

In order to develop our new incremental algorithms, flow network with multiple sinks is needed. Suppose that $N = (V, E, S, T = \{t_1, t_2, \cdots, t_k\}, cap)$, $k \geq 1$, is a single-source *multi-sink* flow network to be solved. In [5], it has been shown that $N$ can be transformed to a *single-sink* network

**Algorithm 1** GENERIC_PREFLOW
$(N = (V, E, S, T, cap))$

1. (INIT) Clear all the flow values and labels to 0. For all $s \in S$, send $cap(s, u)$ units of flow from $s$ to $u$ for all $(s, u) \in E$. Set $d(s) = n$ and $d(v) = 0$ for all sources $s$ and non-sources $v$.
2. Repeatedly perform the following operations until no active vertex exists:
   - Select an active vertex $u$ among the non-source non-sink vertices in $N$.
   - (PUSHRELABEL) Choose and apply one of the following applicable operation on $u$:
     a. PUSH$(v, w)$
     *Applicable*: If vertex $v$ is active, $(v, w)$ is a residual edge, and $d(v) = d(w) + 1$.
     *Action*: Send $min(ex(v), cap(v, w) - f(v, w))$ units of flow from $v$ to $w$.
     b. RELABEL$(v)$
     *Applicable*: If $v$ is active and for every residual edge $(v, w)$, the condition $d(v) \leq d(w)$ holds.
     *Action*: Increase $d(v)$ by letting $d(v) = 1 + \min\{d(w) \mid (v, w)$ is a residual edge$\}$.

[†]For simplicity, this paper does not mention the *skew symmetry* condition which is employed in many related articles.

by adding a *super sink* $\widehat{t}$ and $\infty$-capacity edges to connect each of $t_1, t_2, \cdots, t_k$ to $\widehat{t}$.

In the above transformation, all the flow in original sinks $t_1, t_2, \cdots, t_k$ can be sent to the super sink $\widehat{t}$ via the $\infty$-capacity edge. Conversely, the flow entering the super sink $\widehat{t}$ must be emitted from the original sinks. Thus, clearly, $\sum_{(v,\widehat{t})\in E} f(v,\widehat{t}) = \sum_{(v,t)\in E, t\in\{t_1,t_2,\cdots,t_k\}} f(v,t)$ for *any* flow $f$ on the network. Thus, the total flow enters the sinks $t_1, t_2, \cdots, t_k$ is the same as that enters $\widehat{t}$. In this paper, we do not incur the extra super sink, instead, we simply manipulate $t_1, t_2, \cdots, t_k$ as sink node in a way the same as that in the original Goldberg-Tarjan's preflow algorithm [4].

### 3.3 Abstraction of Generic Preflow Algorithm

In this subsection, we provide a high-level abstraction for the flow network algorithms. The abstraction is helpful on developing the incremental algorithms.

In the generic preflow algorithm, after performing a push/relabel operation, the *state* of the data structures changes, as defined as follows.

**Definition 3.1:** A *state* $q$ with respect to a flow network $N$ is a three tuple $q = (N, f, d)$, where $f$ is a preflow and $d$ is a labeling function. ■

An execution of generic preflow algorithm can be represented as an operation-state sequence

$$\mathcal{I} = (q^0, (op^1, q^1), (op^2, q^2), \cdots, (op^L, q^L)) \tag{1}$$

where $q^i$ is the state immediately after applying $i$-th operation $op^i$, and $L$ is the total number of operations performed. The first $q^0$ and last $q^L$ refer to the state before and after executing all the $L$ operations. Hereinafter, such sequence $\mathcal{I}$ is called a *preflow instance* (or simply *instance*) of generic preflow algorithm. A segment in a preflow instance is called a *sub-instance*. An instance $\mathcal{I} = (q^0, (op^1, q^1), (op^2, q^2), \cdots, (op^L, q^L))$ is called a *prefix* of another instance $\mathcal{I}' = (q'^0, (op'^1, q'^1), (op'^2, q'^2), \cdots, (op'x^{L'}, q'^{L'}))$ if $L \le L'$, and $op^j = op'^j$ for all $j$, $0 \le j \le L$.

The generic preflow algorithm provides us with a very useful guideline: the push/relabel operations can be selected and executed in the arbitrary order without violating the applicable conditions. Lemma 3.1 summarizes this properties.

**Lemma 3.1:** For a flow network $N$, performing INIT function on $N$ leads to a valid state $q$ for $N$. For any active vertex in $q$, there must be one applicable push/relabel operation by executing which another valid state can be obtained and in turn the next active vertex can be selected and manipulated. For generic preflow algorithms, any preflow instance selecting active vertices in the order without violating the applicable conditions is capable of finding the maximum flow.

**Proof.** This lemma holds based on [4].■

### 3.4 Incremental Maximum Flow Algorithm

A sequence of flow networks $(N_1, N_2, \cdots, N_k)$ is *incremental* if $N_i$ and $N_{i+1}$ are the same except that some sink nodes in $N_i$ turn to non-sink in $N_{i+1}$. Formally, assume that $N_i = (V, E, S, T_i, cap)$, then, $(N_1, N_2, \cdots, N_k)$ is a sequence of incremental flow networks if and only if $T_{i+1} \subseteq T_i$ for each $i$, $1 \le i < k$. The *incremental maximum flow problem* is to find the maximum flow for each network $N_1, N_2, \cdots, N_k$.

Hereinafter, let $\setminus$ be the set minus operator. For two sets $S$ and $S'$, define that $S \setminus S' = \{a \mid a \in S \text{ and } a \notin S'\}$. The preflow algorithm selects the active vertices from *non-source non-sink* nodes in the flow network. For $N_i = (V, E, \{s\}, T_i, cap)$, these nodes are called the *active area* and denoted by $A_i = V \setminus \{s \cup T_i\}$. Since $N_i = N_{i+1}$ but $T_{i+1} \subseteq T_i$, we can derive that

$$\begin{aligned} A_{i+1} &= V \setminus \{s \cup T_{i+1}\} \\ &= V \setminus \{s \cup T_i \cup W\} \\ &= A_i \cup W \end{aligned} \tag{2}$$

where $W = T_i \setminus T_{i+1}$ refers to the nodes that are sinks in $N_i$ but turn to non-sinks in $N_{i+1}$.

To illustrate the state transition of the vertices in active areas, let us inspect the active area $A_2$ of network $N_2$ shown in Fig. 3 (b)(c). The set $A_2$ is partitioned as $A_2 = A'_2 \cup A''_2$, where $A'_2 = A_1$ and $A''_2 = A_2 \setminus A_1$. Since $A_1 \subseteq A_2$, the partition $A'_2 \cup A''_2$ is mutually exclusive. When selecting the active vertices, those vertices in $A'_2$ are preferentially considered (see sub-figure (b)). The set $A''_2$ is considered only when no active vertices is presented in $A'_2$ (see sub-figure (c)). Intuitively, since $A'_2 = A_1$, this strategy allows the computations

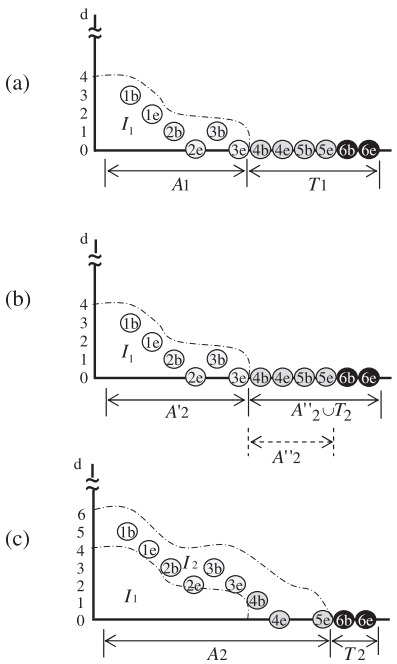



**Fig. 3** Illustration of preflow instances of networks: (a) $N_1$, (b) first stage for $N_2$, and (c) second stage for $N_2$.

in stage 1 (for $N_1$) to be reused in stage 2.

Lemma 3.2 proves that in the incremental maximum flow problem, the computations in stage $i$ can be fully reused in the next stage $i+1$.

**Lemma 3.2:** Assume that $(N_1, N_2, \cdots, N_k)$ is a sequence of single-source multi-sink flow networks. For each pair of consecutive $N_i$ and $N_{i+1}$, $1 \le i < k$, there is an preflow instance $\mathcal{I}$ for $N_i$ and instance $\mathcal{I}'$ for $N_{i+1}$ such that $\mathcal{I}$ is a prefix of $\mathcal{I}'$.

**Proof.** Let $N_i = (V, E, \{s\}, T_i, cap)$ and $N_{i+1} = (V, E, \{s\}, T_{i+1}, cap)$. Assume that $\mathcal{I} = (q^0, (op^1, q^1), (op^2, q^2), \cdots, (op^L, q^L))$ is a preflow instance that finds the maximum flow on $N_i$. Based on $\mathcal{I}$, we can construct the following sub-instance

$$\mathcal{I}' = (q'^0, (op^1, q'^1), (op^2, q'^2), \cdots, (op^L, q'^L))$$

for $N_{i+1}$, where for each $j$, operation $op^j$ is the same as that in $\mathcal{I}$, and state $q'^j = (N_{i+1}, f, d)$ is the same as $q^j = (N_i, f, d)$ in $\mathcal{I}$ but change $N_i$ to $N_{i+1}$.

According to Eq. (2), $N_{i+1}$ has larger active area than $N_i$, *i.e.* $A_{i+1} = A_i \cup W$. Partition $A_{i+1}$ into $A_{i+1} = A'_{i+1} \cup A''_{i+1}$, where $A'_{i+1} = A_i$ and $A''_{i+1} = W$. The partition is mutually exclusive because $A_i \subset A_{i+1}$. Since $N_{i+1}$ and $N_i$ have exactly the same graph structure and edge costs, the fact that $\mathcal{I}$ is a preflow instance (for $N_i$) manipulating active vertices in $A_i$ implies that $\mathcal{I}'$ is a valid preflow sub-instance manipulating the vertices in $A'_{i+1} = A_i$ for $N_{i+1}$.

From Lemma 3.1, executing the sub-instance $\mathcal{I}'$ leads to a valid state $q'^L$ on $N_{i+1}$. Starting from state $q'^L$, we can apply push/relabel operations, until no active vertices can be found. Let

$$\mathcal{I}'' = (q'^L, (op^{L+1}, q'^{L+1}), (op^{L+2}, q'^{L+2}), \cdots, (op^M, q'^M))$$

be the sequence of operations applied after $q'^L$. The concatenance of $\mathcal{I}'$ and $\mathcal{I}''$ forms a preflow instance for network $N_{i+1}$. Also, $\mathcal{I}$ is a prefix of the concatenated sequence because $\mathcal{I}'$ and $\mathcal{I}$ have the same operation sequence. Thus the lemma follows.■

Based on Lemma 3.2, the operation sequence of stage $i$ can be a prefix of that of next stage $i+1$. This confirms the correctness of Algorithm 2, in which (line 6) the initial state of stage $i+1$ is directly cloned from the final state of stage $i$. With this incremental approach, the total time to solve the $k$ problems over $N_1, N_2, \cdots, N_k$ is the same as that to solve the last problem over $N_k$. As shown in [4], the generic preflow runs in $O(n^2m)$ time, where $n$ and $m$ are the number of vertices and edges, respectively.

**Algorithm 2** INCREMENTAL_PREFLOW $(N_1, N_2, \cdots, N_k)$

1: Let $N_i = (V, E, \{s\}, T_i, cap)$. If $N_1, N_2, \cdots, N_k$ are not incremental flow networks, return an error.
2: **for** stage $i = 1$ to $k$ **do**
3: **if** $i = 1$ **then**
4: Let state $q_1^0 = \{N_1, f, d\}$ be the state right after invoking function INIT() on $N_1$ (given in Algorithm 1).
5: **else**
6: Let $q_i^0 = q_{i-1}^{(last)}$ but change $N_{i-1}$ to $N_i$
7: **end if**
8: Invoke function PUSHRELABEL() on state $q_i^0$ (given in Algorithm 1).
9: Let $q_i^{(last)} = \{N_i, f, d\}$ be the final state after invoking PUSHRELABEL().
10: **end for**
11: **return** the maximum flow values of final states $q_i^{(last)}$ of all stages $i = 1, 2, \cdots, k$.

## 4. Content Deployment Algorithms

### 4.1 Basic Content Deployment Algorithm

The basic content deployment problem is to find an optimal way to delivery a set of target objects. When deploying, some objects are *useless* and can be disregarded to speedup the calculation. For example, observing the ODG in Fig. 4, if $J = \{4, 5\}$ is the target then each object in $\{6, 7, 8\}$ is neither itself a target object nor an upstream of any target object. Clearly, such objects can be disregarded when deploying the target objects. Formally, in an ODG $G = (U, F, I, J, net, comp)$, the target objects as well as their upstream objects (*i.e.* $\mathcal{P}(J) \cup J$) are called *useful objects*, and those downstream objects of $J$ (*i.e.* $\mathcal{S}(J)$) are called *useless objects*.

A basic content object deployment problem can be solved by reducing it to the flow network problem. The reduction is described in Algorithm 3 and explained below. Note that the reduction is simplified from that in [1] but with some extension for the incremental properties.

An illustration of the reduction is shown in Fig. 4. Let

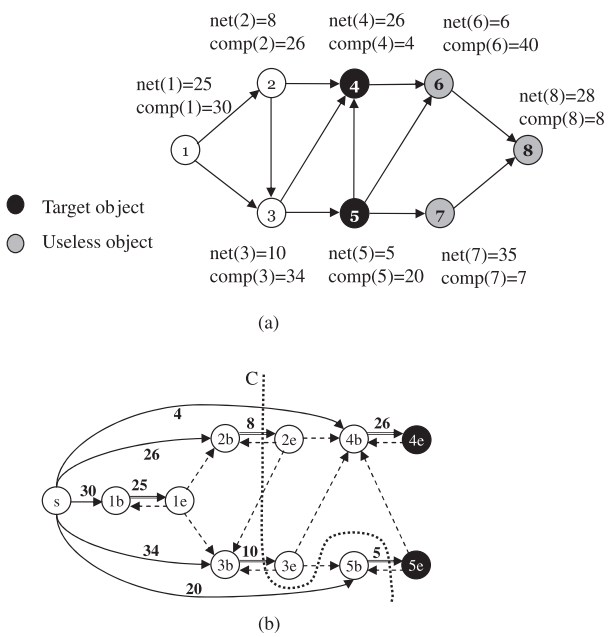



**Fig. 4** (a) An object dependency graph $G$. (b) The reduced flow network $N$. Note that the useless objects in $G$ are not transformed in $N$.

**Algorithm 3** BASIC_CONTENT_DEPLOYMENT($G = (U, F, I, J, net, comp)$)

1. (**Reduction**) Perform the following vertex-to-edge transformation from ODG $G = (U, F, I, J, net, comp)$ to flow network $N = (V, E, S, T, cap)$:
   a. For each object $a$ in $G$, add two nodes $a_{begin}$ and $a_{end}$, and two edges $(a_{begin}, a_{end})$ and $(a_{end}, a_{begin})$ to $N$. Let $cap(a_{begin}, a_{end}) = net(a)$ and $cap(a_{end}, a_{begin}) = \infty$.
   b. For each precedence relation $(a, b)$ in $G$, add an $(a_{end}, b_{begin})$ in $N$ and let $cap(a_{end}, b_{begin}) = \infty$.
   c. Add a source node $s$ to $N$. For each object $a$, add edge $(s, a_{begin})$ to $N$. Let $cap(s, a_{begin}) = comp(a)$.
   d. Set the set of sink nodes $T = \{a_{end} \mid a \in J\} \cup \{a_{begin}, a_{end} \mid a \in \mathcal{S}(J)\}$.
2. After the flow network $N$ is constructed, find the minimum cut $C = (X, Y)$ of $N$ by using the existing maximum flow algorithms.
3. (**Mapping**) Construct the minimum deployment $\mathcal{D} = (U_O, U_N, U_C)$ based on the following mapping:
   - Object $a \in U_O \Leftrightarrow (a_{begin} \in X$ and $a_{end} \in X$ in $N)$
   - Object $a \in U_N \Leftrightarrow (a_{begin} \in X$ and $a_{end} \in Y$ in $N)$
   - Object $a \in U_C \Leftrightarrow (a_{begin} \in Y$ and $a_{end} \in Y$ in $N)$
4. return $\mathcal{D}$

$G = (U, F, I, J, net, comp)$ be an ODG and let $N$ be the flow network reduced from $G$ by invoking Algorithm 3. Hereinafter, a cut in $N$ without cutting any $\infty$-cost edges is called a *feasible cut*. Clearly, a minimum cut must be feasible.

For each pair of nodes $(a_{begin}, a_{end})$ in $N$, there are only three cases the nodes are partitioned by a *feasible* cut $C = (X, Y)$, as described follows:

**M1** ($a_{begin} \in X$ and $a_{end} \in X$): For this case, $a \in U_O$ and no cost is contributed to the cut cost.

**M2** ($a_{begin} \in X$ and $a_{end} \in Y$): For this case, $a \in U_N$ and the value $cap(a_{begin}, a_{end}) = net(a)$ is contributed to the cut cost.

**M3** ($a_{begin} \in Y$ and $a_{end} \in Y$): For this case, $a \in U_C$ and the value $cap(s, a_{begin}) = comp(a)$ is contributed to the cut cost.

Note that the case ($a_{begin} \in Y$ and $a_{end} \in X$) can not happen since if so the reverse edge $(a_{end}, a_{begin})$, with capacity $\infty$, will be cut, which contradicts to that $C$ is feasible.

An example of rules M1-M3 is illustrated in Fig. 4. In sub-figure (a), objects $2, 3, 5$ are in $U_N$, based on rule M2, their corresponding edges $(2_b, 2_e), (3_b, 3_e), (5_b, 5_e)$ are cut by $C$ in figure (b). Thus, cost $net(2) + net(3) + net(5)$ is added to the cut cost. Also, objects 4 are in $U_C$, based on rule M3, the corresponding nodes $4_b, 4_e$ are located in the right-hand-side of cut $C$. Thus, cost $comp(4)$ is added to the cut cost. The useless objects $\{6, 7, 8\}$ are disregarded.

**Lemma 4.1:** Let $G$ be an ODG without useless objects, and let $N$ be the reduced flow network of $G$. Then, $cost(MinC(N)) = cost(MinD(G))$.

**Proof.** The condition holds based on the mapping rules M1-M3 [1]. The details of proof are ignored in this paper.■

### 4.2 Incremental Content Deployment Algorithm

Now we are ready to discuss our incremental content deployment algorithm, which is simply a composition of the non-incremental algorithm. The incremental algorithm is described as follows:

1. Reduce the incremental ODGs $(G_1, G_2, \cdots, G_k)$ to the incremental flow networks $(N_1, N_2, \cdots, N_k)$. Note that $G_i$ may have useless objects.
2. Find the minimum cut $C_i$ of $N_i$, $i = 1, 2, \cdots, k$, by invoking Algorithm 2. Then, construct deployment $\mathcal{D}_i$ from $C_i$ based on mapping rules M1-M3.
3. (**PostProcessing**) For each stage $i = 1, 2, \cdots, k$, after performing Step 2, every useless object is assigned to the set $U_C$ in $\mathcal{D}_i$ (according to rule M3). However, useless objects are supposed to be non-deployed. In order to obtain a correct solution, directly moving the useless objects from $U_C$ to $U_O$ in $\mathcal{D}_i$.
4. return $(\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_k)$.

Different with the previous non-incremental algorithm which simply disregards useless objects (see Lemma 4.1), the useless objects are invloved in the above incremental algorithm.

When reducing ODG $G_i$ to the flow network $N_i$, each object $a$ in $G_i$ is transformed to two vertices $\{a_{begin}, a_{end}\}$ in $N_i$. For an object set $B$ in $G_i$, let $\widehat{V}_{begin}(B) = \{a_{begin} \mid a \in B\}$ and $\widehat{V}_{end}(B) = \{a_{end} \mid a \in B\}$ be the vertices in $N_i$ transformed from objects in $B$. Also, let $\widehat{V}(B) = \widehat{V}_{begin}(B) \cup \widehat{V}_{end}(B)$. Recall that in the ODG $G_i$, $\mathcal{P}(J_i) \cup J_i$ refers to the useful objects while $\mathcal{S}(J_i)$ refers to the useless objects. Mapping to the flow network $N_i = (V, E, \{s\}, T_i, cap)$, $V = \{s\} \cup A_i \cup T_i$ where $s$ is the source node, and

$$A_i = \widehat{V}(\mathcal{P}(J_i)) \cup \widehat{V}_{begin}(J_i) \qquad (3)$$
$$T_i = \widehat{V}_{end}(J_i) \cup \widehat{V}(\mathcal{S}(J_i))$$

Figure 5 gives an intuitive illustration. In the figure, the targets for ODGs $G_2$ and $G_3$, are $J_2 = \{4, 5\}$ and $J_3 = \{6, 7\}$. Also, useless objects are $\mathcal{S}(J_2) = \{6, 7, 8\}$ and $\mathcal{S}(J_3) = \{8\}$. Note that useless objects $\{6, 7\}$ in $G_2$ turn to useful in $G_3$. Mapping to the flow networks, $T_2 = \{4e, 5e\} \cup \{6b, 6e, 7b, 7e, 8b, 8e\}$ and $T_3 = \{6e, 7e\} \cup \{8b, 8e\}$. This implies that sinks $\{4e, 5e, 6b, 7b\}$ in $N_2$ turns to non-sink in $N_3$ and confirms that $N_2$ and $N_3$ are incremental.

Lemma 4.2 proves the above property formally.

**Lemma 4.2:** Consider a sequence of incremental ODGs $(G_1, G_2, \cdots, G_k)$. Let $N_i$ be the reduced flow network of $G_i$. Then, $(N_1, N_2, \cdots, N_k)$ is a set of incremental flow networks.

**Proof.** Consider two consecutive ODGs $G_i = (U, F, I, J_i, net, comp)$ and $G_{i+1} = (U, F, I, J_{i+1}, net, comp)$. Since $G_i$ and $G_{i+1}$ are strictly incremental, the condition $J_i \prec J_{i+1}$ holds and implies that $\mathcal{S}(J_{i+1}) \subset \mathcal{S}(J_i)$ and $\mathcal{P}(J_i) \subset \mathcal{P}(J_{i+1})$. This further implies that $\{J_{i+1} \cup \mathcal{S}(J_{i+1})\} \subset$

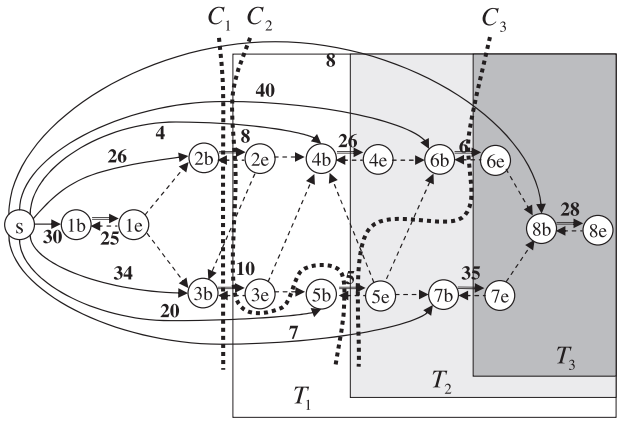



**Fig. 5** Incremental flow networks $N_1, N_2, N_3$ that are reduced from ODG in Fig. 4 (a) with target object $J_1 = \{2, 3\}$, $J_2 = \{4, 5\}$, and $J_3 = \{6, 7\}$. For each $N_i$, its sink nodes is $T_i = \{a_{begin} \mid a \in J_i\} \cup \{a_{begin}, a_{end} \mid a \in \mathcal{S}(J_i)\}$.

$\{J_i \cup \mathcal{S}(J_i)\}$ (see Fig. 2 and Fig. 5). Therefore, the incremental property of $(N_1, N_2, \cdots, N_k)$ can be established based on Eq. (3):

$$\begin{aligned} &\Rightarrow \quad \{J_{i+1} \cup \mathcal{S}(J_{i+1})\} \subset \{J_i \cup \mathcal{S}(J_i)\} \\ &\Rightarrow \quad \{\widehat{V}_{end}(J_{i+1}) \cup \widehat{V}(\mathcal{S}(J_{i+1}))\} \\ &\qquad\quad \subset \{\widehat{V}_{end}(J_i) \cup \widehat{V}(\mathcal{S}(J_i))\} \\ &\Rightarrow \quad T_{i+1} \subset T_i \end{aligned} \tag{4}$$

■

Next, Lemma 4.3 shows the correctness of above incremental algorithm.

**Lemma 4.3:** Consider a sequence of strictly incremental ODGs $(G_1, G_2, \cdots, G_k)$, where $G_i = (U, F, I, J_i, net, comp)$. Let $N_i$ be the reduced flow network of $G_i$. Algorithm 3 finds the minimum deployment of $G_i$ with the following property:

- $cost(MinC(N_i)) = cost(MinD(G_i)) + h_i$, where $h_i = \sum_{a \in \mathcal{S}(J_i)} comp(a)$, $i = 1, 2, \cdots, k$.

**Proof.** Assume that $G_i = (U, F, I, J_i, comp, net)$ and $N_i = (V, E, \{s\}, T_i, cap)$. Let $G'_i$ be the ODG the same as $G_i$ but excluding the useless objects, and let $N'_i$ be the reduced flow network of $G'_i$. Analogously, $N'_i$ is the same as $N_i$ but excluding the vertices corresponding to the useless objects and their incident edges. Based on Lemma 4.1, the following condition holds

$$cost(MinC(N'_i)) = cost(MinD(G'_i)) \tag{5}$$

The main task of this proof is to examine the effect of involving useless objects in $G'_i$ and $N'_i$.

An edge in $N_i$ is called a *trivial edge* if it directly connects the *source* and a *sink* in the network. Clearly, the following property holds for this kind of edges:

**P1** Every cut of $N_i$ must cut all the trivial edges. Consequently, the minimum cut value of $N_i$ is equal to the minimum cut value of $N'$ plus value $h$, where $N'$ is a flow network the same as $N_i$ but *excluding* the trivial edges, and $h$ is the sum of costs of trivial edges.

In Algorithm 3, for each useless object $a \in \mathcal{S}(J_i)$ in $G_i$, there are two nodes $\{a_{begin}, a_{end}\}$ and a trivial edge $(s, a_{begin})$ in $N_i$. Note that $\{a_{begin}, a_{end}\} \subseteq \widehat{V}(\mathcal{S}(J_i)) \subseteq T_i$. Since the ODG $G_i$ is *strictly* incremental, there is no link from object in $\mathcal{P}(J_i)$ to object in $\mathcal{S}(J_i)$ in the ODG. Mapping to $N_i$, there is *no* edge connect $\widehat{V}(\mathcal{P}(J_i))$ and $\widehat{V}(\mathcal{S}(J_i))$. Therefore, in $N_i$, the edges connect to $\widehat{V}(\mathcal{S}(J_i))$ must be start from source $s$ (those edges between sink nodes are disregarded), as listed in Step 1c in Algorithm 3. The following property summarizes the above properties:

**P2** For each useless object $a \in \mathcal{S}(J_i)$ in $G_i$, there is a trivial edge $(s, a_{begin})$ with cost $cap(s, a_{begin}) = comp(a)$ in $N_i$. Conversely, each trivial edge $(s, a_{begin})$ in $N_i$ implies that $a$ is an useless object in $G_i$. The total cost of trivial edges is equal to $\sum_{a \in \mathcal{S}(J)} comp(a)$.

For example, in Fig. 5, if the target $J_2 = \{4, 5\}$ and useless objects are $\mathcal{S}(J_2) = \{6, 7, 8\}$, the corresponding nodes $\widehat{V}(\mathcal{S}(J_2))$ is $\{6b, 6e, 7b, 7e, 8b, 8e\}$, and the trivial edges are $(s, 6b), (s, 7b), (s, 8b)$.

According to Property P1 and P2, we can derive that the minimum-cut cost of $N_i$ is equal to the minimum-cut cost of $N'_i$ plus the total cost of trivial edges in $N_i$. Thus, $cost(MinC(N_i)) = cost(MinC(N'_i)) + h$, where $h = \sum_{a \in \mathcal{S}(J)} comp(a)$. Combined with Eq. (5), this yields the property $cost(MinC(N_i)) = cost(MinD(G'_i)) + h$.■

## 5. Experimental Results

In this section we discuss the experimental results of applying our incremental technique to the flow network problems. Our experiments aim to examine the correctness and performance of the incremental techniques for various types of graphs. In the experimentation, the sets of weighted directed acyclic graphs, each with 10000 nodes, are generated randomly, based on following rules. Each node in a graph is numbered with an unique id. The initial node is number with 0. The edges are added to the graph iteratively. When adding a new edge $(u, v)$, the start node $u$ is chosen randomly from those nodes reachable form the initial node, and the end node $v$ is chosen from those nodes with id larger than $u$'s id. By employing this rule we obtain acyclic and connected graphs.

In an ODG, an object $a$ with $net(a) \leq comp(a)$ is called a *predetermined* object. Clearly, predetermined objects must be deployed via directly network transmission regardless of the status of other objects. Let $\alpha$ be the ratio of the number of predetermined objects to the total number of objects in ODGs. We examine how $\alpha$ value affects the performance of the incremental algorithm. In the experimentation, three $\alpha$ values (25%, 50%, and 75%) are tested. Figure 6 plots the experimental results in terms of the incremental technique and the ratio of predetermined objects in the ODG. In the figure, the Y axis represents the execution time of the program, while the X axis represents the stage number of the incremental algorithm. Each measured value

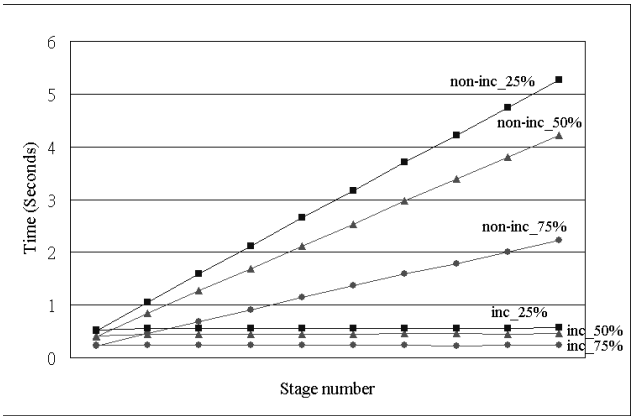



**Fig. 6** Experimental results for $\alpha = 25\%$, $\alpha = 50\%$, and $\alpha = 75\%$, where $\alpha$ is the ratio of the number of predetermined objects to the total number of objects in ODG. For each $\alpha$ value, a pair of curves for the results with and without the incremental technique are plotted and marked with 'inc' and 'non-inc', respectively.

on Y axis is obtained by taking an average on 100 execution results. As shown in Fig. 6, the incremental technique obviously improves the performance since the operations in the earlier stages can all be reused. The experimental result also shows that the incremental technique gains greater benefits from ODGs with low $\alpha$ value. The effect of incremental technique degenerates as $\alpha$ increases because high $\alpha$ value indicates high ratio of predetermined objects in ODG and no combinatorial optimization calculation is required for predetermined objects (they have only one optimal deploying policy *i.e.* directly network transmission).

## 6. Discussion

In this paper, the maximum flow problem for a single graph is extended to an incremental growing graph. This work shows that the time complexity of deriving $k$ maximum flow values of incremental graphs $N_1, N_2, \cdots, N_k$ is no more than that of the single graph $N_k$. Based on this result, this work develops the incremental content object deployment algorithm. The experimental results show that our incremental techniques dramatically improve the performance of the object deployment algorithm. In this paper, the incremental technique only works for strictly incremental graphs with static costs, extending the technique to non-strictly incremental graphs and dynamic costs will be the future works.

## Acknowledgments

The authors would like to acknowledge the anonymous referees for their valuable comments. This work was supported by contracts from National Science Council, Taiwan, ROC, under Grant NSC-97-2221-E-029-022.

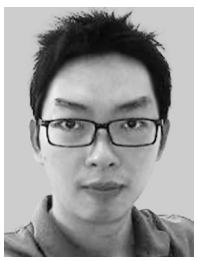
### References

[1] X. Tang and S. Chanson, "Minimal cost replication of dynamic web contents under flat update delivery," IEEE Trans. Parallel Distrib. Syst., vol.15, no.5, pp.431–441, May 2004.
[2] K.H.Y.L. Chin and W. Zhang, "Multimedia object placement for transparent data replication," IEEE Trans. Parallel Distrib. Syst., vol.18, no.2, pp.212–224, 2007.
[3] L. Ford and D. Fulkerson, "Maximal flow through a network," Can. J. Math., vol.8, pp.399–404, 1956.
[4] A. Goldberg and R. Tarjan, "A new approach to the maximum-flow problem," J. ACM, vol.35, no.4, pp.921–940, Oct. 1988.
[5] C.L.T.H. Cormen and R. Rivest, Introduction to Algorithms, The MIT Press, 1989.
[6] R.A.T. Magnanti and J. Orlin, Network Flows: Theory, Algorithms, and Applications, Prentice-Hall, 1993.
[7] K.H.K. Nakano and M. Sengoku, "The p-collection problem in a flow network with lower bounds," IEICE Trans. Fundamentals, vol.E80-A, no.4, pp.651–657, April 1997.
[8] A.S. Fujishige and T. Takabatake, "A polynomial-time algorithm for the generalized independent-flow problem," J. Oper. Res. Soc. Jpn., vol.47, pp.1–17, 2004.
[9] L. Chen and I. Wu, "Detection of summative global predicates," IEICE Trans. Inf. & Syst., vol.E86-D, no.5, pp.976–980, May 2003.
[10] I. Wu and L. Chen, "On detection of bounded global predicates," Comput. J., vol.41, no.4, May 1998.

**Lung-Pin Chen** is an associate professor of Department of computer science and information engineering at Tung-Hai University, Taiwan. He received his B.S. from Soochow University in September 1991, M.S. from National Chung-Cheng University in September 1993, and Ph.D. from National Chiao-Tung University in January 1999, all in computer science. His research interests include internet/network computing, XML database, and pervasive computing.

**I-Chen Wu** is a professor in the Department of Computer Science, at National Chiao Tung University (NCTU). He received his B.S. in Electronic Engineering from National Taiwan University (NTU), M.S. in Computer Science from NTU, and Ph.D. in Computer Science from Carnegie-Mellon University, in 1982, 1984 and 1993, respectively. Dr. Wu developed a game platform over Internet and helped start up a software development company, ThinkNewIdea Inc in Taiwan, in 2000. Dr. Wu also invented a new game, named Connect6, a variation of the five-in-a-row game in 2005. He has been the director of IGS-NCTU Joint Research Center at NCTU and the president of Taiwan Connect6 Association since 2007.

**William Chu** is a professor of Department of computer science and information engineering at Tung-Hai University, Taiwan. He received his B.S. in Electronic Engineering from Tamkang, M.S. and Ph.D. in Computer Science from Northwestern University, USA, in 1984 and 1992, respectively. His research interests object-oriented, software engineering, reengineering, software maintenance, software agent, E-commerce, and design patterns. Dr. Chu is currently the Dean of School of Engineering of Tung-Hai University.



**Jhen-You Hong** received the M.S. degree in Computer Science and Information Engineering from Tunghai University, Tai-Chung, Taiwan in 2009. His research interests include distributed systems, contents management, and replica management.

**Meng-Yuan Ho** received the M.S. degree in Computer Science and Information Engineering from Tunghai University, Tai-Chung, Taiwan in 2009. His research interests include distributed systems, and testing and debugging distributed programs.