Vol. 41 No. 2 Transactions of Information Processing Society of Japan Feb. 2000

*Regular Paper*

# An Efficient Distributed Algorithm for Implementation of Multi-Rendezvous based on *l*-Chain-Coterie

Zixue Cheng,[†1] Kshirasagar Naik,[†2] Naka Tajima,[†3]
Tongjun Huang[†4] and Shoichi Noguchi[†1]

Multi-Rendezvous is a powerful communication mechanism that allows a group of processes to execute an event in a synchronous way. Besides the multi-synchronous property, Multi-Rendezvous has an exclusive property which requires no more than one group to execute their events simultaneously, if they share a common process. It is not a trivial task to implement Multi-Rendezvous in a network system. The implementation has to maintain the properties mentioned above, be fair and make progress. Some highly abstract specification languages have employed Multi-Rendezvous to enhance the specification power of the languages, e.g., LOTOS, a specification language for communication protocols and distributed systems. *Coterie* is a communication structure which has been used in solutions for many distributed problems such as mutual exclusion, replica control, and distributed consensus. Our previous work has shown that *coterie* can also be used for solving implementation problem of Multi-Rendezvous, with $O(N^{1.5})$ message passes. In this paper, we propose a communication structure called $l$-chain-coterie, as a generalization of *coterie*, and a more efficient distributed algorithm for implementing Multi-Rendezvous based on the structure with $O(N^{1+1/l})$ message passes. Our algorithm is fully distributed and does not use manager processes and auxiliary resources.

## 1. Introduction

Multi-Rendezvous is a communication mechanism which extends binary rendezvous that has been suggested for CSP and Ada. Multi-Rendezvous has *Multi-Synchronous* and *Exclusive* properties. Multi-Synchronization means that a group of processes interact with each other to participate in the execution of an event at the same time. Generally, there are more than one group of processes in a distributed system. Exclusion means that no more than one group execute their events simultaneously, if they share a common process.

Some highly abstract specification languages have employed Multi-Rendezvous to enhance the specification power of the languages. For example, LOTOS [1], a high level specification language for OSI communication protocols, services, and other distributed systems, incorporates Multi-Rendezvous as its communication mechanism. Proposals for distributed programming such as Script [13] and Interacting Processes (IP) [14] also employ a Multi-Rendezvous mechanism.

To really benefit from the Multi-Rendezvous, Multi-Rendezvous has to be implemented in a network environment which only has a one-to-one message passing mechanism. The implementation of Multi-Rendezvous in a network environment is not a trivial task. The problem is to design a distributed algorithm that decides which group of processes should execute their event in order to guarantee the multi-synchronous and exclusive properties. The decision made by the distributed algorithm has to be fair and make progress. Fairness means if a group of processes try to participate in an enabled event infinitely often, the event will be eventually executed☆. Progress means when an event is enabled, either the event or some conflicting event will eventually be executed.

There are several works on implementation of Multi-Rendezvous in a network environment. The distributed algorithm proposed by Gao and Bochmann requires $O(N^2)$ message passes [2]. The solution by Sisto et al. requires only $O(N)$ massages [3]. However it uses a binary tree structure and messages are sent to the root of the tree for making a decision. Therefore this approach is not fully distributed.

Chandy and Misra first pointed out that

†1 Department of Computer Software, University of Aizu
†2 School of Computer Science, Carleton University
†3 SOGO KEIBI HOSHO CO., LTD.
†4 Information System and Technology Center, University of Aizu

☆ For the definition of enabled event, see Section 2.

the problem of implementing Multi-Rendezvous is associated with synchronization and exclusion problems [4]. They described the Multi-Rendezvous as a committee coordination problem and proposed an elegant algorithm to implement it. In their algorithm, auxiliary resources are used to implement synchronization and exclusion. That is, for every event in the system, a special process called a coordinator is postulated and a fork is used between two neighbor groups of processes [4].

To improve Chandy and Misra's algorithm, Bagrodia proposed an event manager algorithm [5),6)] to avoid using the forks. However, the event manager algorithm uses a circulating token to solve the exclusion problem. The token is circulated around all event managers sequentially, so it is not a completely distributed one.

In this paper, we propose a distributed algorithm with $O(N^{1+1/l})$ message passes, where $N$ is the number of processes in the network and $l$ is an integer not less than 2. Our algorithm is completely distributed and it does not use manager processes and auxiliary resources. We propose a communication structure called $l$-chain-coterie, which is a generalization of *coterie*. The proposed algorithm is based on the $l$-chain-coterie and a *counter*-based method to solve the implementation problem of Multi-Rendezvous.

Coteries are a kind of communication structure which have been used in many distributed problems, such as mutual exclusion [7)], replica control [8)], and distributed consensus [9)]. Coteries can also be used for solving the implementation problem of Multi-Rendezvous (for example see Cheng, et al.[16)]). The the lower bound of message complexity for implementation of Multi-Rendezvous based on coteries is $O(N^{1.5})$, because processes have to achieve consensus on their local synchronous conditions to maintain the multi-synchronization property, and lower bound for achieving consensus based on coteries is $O(N^{1.5})$ [9)], under the condition that the load and responsibility of every process is the same. The $l$-chain-coterie, proposed in this paper, provides a more efficient solution, whose message complexity is $O(N^{(1+1/l)})$. When $l = 2$, the $l$-chain coterie is reduced to *coterie*.

$k$-coterie is another kind of generalization of *coterie* proposed for solving the $k$-mutual exclusion problem effectively [11),12)]. $k$-coterie is not suitable for implementation of Multi-Rendezvous. This is because every process in a process-set has to send (directly or indirectly) its local synchronous conditions to all other processes in the process-set to maintain the multi-synchronization. The intersection property of $k$-coterie cannot guarantee there is a communication path between an arbitrary pair of processes. Recently, weighted $k$-quorum systems are proposed for solving $k$-mutual exclusion problem [15)]. The $k$-quorum systems can also be used to solve the implementation problem of Multi-Rendezvous, since the intersection of any pair of quorums is not empty and a pair of processes can communicate with each other. Because the $k$-quorum systems are quorum systems, solutions based on it are as efficient as coteries.

Section 2 gives a precise description of the problem and the definition of $l$-chain-coterie. Section 3 presents our solution to the problem. In Section 4, properties and complexity of the distributed algorithm are given. Section 5 contains our conclusions.

## 2. Network Model and Problem Description

### 2.1 Network Model

In this paper, we use a connected and undirected graph, to represent a network, with $N$ processes and a link between each pair of processes. There is no centralized controller, shared memory, or global clock in the network. The processes communicate with each other only by exchanging messages through the links. Messages sent by a process over a link will arrive at another process in the sequence as they are sent, after an unpredictable but finite delay. We assume that messages can be transmitted independently in both directions on a link and there is no error with respect to processes and links in the network.

### 2.2 The Problem

Roughly speaking, Multi-Rendezvous is a communication mechanism which considers a set of processes participating in a set of events.

For the formal definition of the Multi-Rendezvous problem, we follow Bagrodia's approach [5)] basicly. We denote the set of processes in the network by $\mathbf{P} = \{p_1, p_2, \cdots, p_i, \cdots, p_n\}$ and the set of events by $\mathbf{E} = \{e_1, e_2, \cdots, e_k, \cdots, e_m\}$. Each process may participate in an event of a subset of $\mathbf{E}$. The set $E_i \subseteq \mathbf{E}$ of events, which $p_i$ may participate in, is called $p_i$'s *event-set*. Each event is partic-

**Table 1** Synchronous conditions.

| $p_i$ | $p_j$ | Synchronous conditions | Interaction kinds |
|---|---|---|---|
| $e!V_1$ | $e!V_2$ | value($V_1$) = value($V_2$) | value matching |
| $e!V$ | $e?x:t$ | value($V$) in domain($t$) | value passing |
| $e?x:t$ | $e?y:u$ | $t = u$ | value generation |

ipated by some processes. The set $P_k \subseteq \mathbf{P}$ of processes which participate in event $e_k$ is called $e_k$'s *process-set*. Events $e_k$ and $e_j$ conflict with each other, if their process-sets have at least one common process, i.e., $P_k \cap P_j \neq \phi$. We assume, in this paper, that the elements of $E_i$ and $P_k$ don't change during the execution of the algorithm.

A process $p_i$ is in either idle or active state. In active state, $p_i$ is participating in an event $e_k$ or is doing some local work independent of other processes. $p_i$ is said to be active for an event $e_k$ or some local work, if it is participating in $e_k$ or is doing the local work. In idle state, $p_i$ waits for participating in any one of the events in $E_i$. Process $p_i$ participates in an event $e_k$ only when all other processes in $P_k$ will also participate in $e_k$ (synchronous property). An idle process may participate in at most one event at any time.

A process $p_i$ autonomously changes its state from the active one to the idle one. However, $p_i$ changes its state from the idle one to the active one, only if it participates in some event. Otherwise, it remains idle.

An event is enabled, disabled, or executed. An event $e_k$ is enabled if and only if all processes in $P_k$ are idle and these processes satisfy synchronous conditions; $e_k$ is disabled if there exists some $p_i \in P_k$ such that $p_i$ is activefor an event $e_j$ $(j \neq k)$ or synchronous conditions on the event are not satisfied. Intuitively, an enabled event $e_k$ means that every process in $P_k$ is waiting for others and conditions on data types and values exchanged by these processes in the communication are agreed with each other.

The specific synchronous conditions depend on languages which incorporate Multi-Rendezvous. For example in LOTOS, there are three kinds of synchronous conditions as shown in **Table 1** (see Ref. 1) for detail). In this paper, we use the denotations of synchronous conditions in the table. Our method can be applied to other languages with simple modification.

For any pair of $p_i$, $p_j \in P_k$ the above synchronous conditions need to be satisfied in order for $e_k$ to be enabled. Here $e!V$ is a structured event. "$e$" is an event name. "$!V$" is a parameter of the event, representing a value. Similarly, "$?x:t$" is a parameter representing a variable $x$ and its type $t$.

An event $e_k$ is executed if and only if each process in $P_k$ has decided to participate in $e_k$. Note that an enabled event $e_k$ may not be executed, since a process $p_i$ in $P_k$ may have more than one enabled event in its event-set $E_i$, and $p_i$ may decide to participate in another enabled event $e_l$.

Our task is to devise an algorithm which arranges an idle process to participate in an enabled event, with the following properties being satisfied.

**Synchronous property**[☆]**:** A disabled event can't be executed.

**Exclusive property:** Events in conflict can't be executed simultaneously.

**Fairness property:** If an event becomes enabled infinitely often, the event will be eventually executed.

**Progress property:** If there is an enabled event, either the event or at least one of its conflicting events will be executed eventually.

## 3. The Distributed Algorithm

### 3.1 Communication Structure

#### 3.1.1 *l*-chain-coterie

In this paper, we use $l$-chain-coterie as communication structure, for solving efficiently the implementation problem of Multi-Rendezvous, specially for maintaining the multi-synchronous property.

A naive method for maintaining the multi-synchronous property is that each process sends its local synchronous conditions to all other processes in the same process-set and each checks whether the conditions are matched with others. This method requires $O(N^2)$ messages and one round of message exchanges.

A coterie-based method is as follows[☆☆]. Every process sends its local conditions to processes in some quorums, whose size (the number of processes in the quorums) is $O(N^{0.5})$. On receiving messages carrying the local conditions, each process checks whether they are matched

☆ We use "synchronous" to mean "multi-synchronous" hereafter.

☆☆ A coterie is a family of subsets (called quorums) of processes, such that every pair of subsets have at least one common process, and no subset contains another subset.

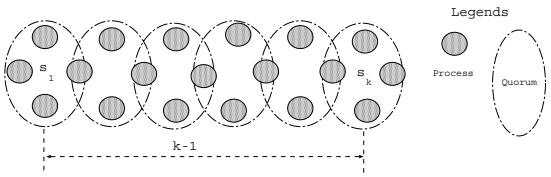


**Fig. 1** Intuitive view of $l$-chain-coterie.

or not, and lets the processes in the quorums know the checking result. Although two rounds of message exchanges are required, in order for all processes to know whether their conditions are matched or not, this method reduces the message complexity to $O(N^{1.5})$, under the condition that the coterie is symmetric.

Our purpose is to further reduce the message complexity by reducing the size of quorums, though more rounds of message exchanges are required.

**Definition 1 ($l$-chain-coterie)**

Let $P$ be a set of processes. A $l$-chain-coterie (abbreviated to $LCC$) is a family of nonempty subsets of $P$, such that both the following properties are satisfied.

(1) *Intersection property*: Let $k$ be an integer such that $1 < k \leq l$. For every pair $s_1, s_k \in LCC$, there are $s_i \in LCC\,(1 < i < k)$, such that $s_1 \cap s_2 \neq \phi, \cdots, s_{i-1} \cap s_i \neq \phi, \cdots, s_{k-1} \cap s_k \neq \phi$.

(2) *Minimality property*: $\forall s_i, s_j \in LCC$, $s_i \nsubseteq s_j$ and $s_j \nsubseteq s_i$.

An element of $LCC$ is called a quorum. □

Intuitively, there is a chain of quorums for any pair of quorums, as shown in **Fig. 1**. If we consider every quorum is a vertex and there is edge between a pair of quorums which have a common process, The maximum length of a chain in a $l$-chain-coterie is $l - 1$.

**Lemma 1** A message sent by a process $p_i$ can arrive to another process $p_j$, by at most $l$ rounds of sending and receiving messages, using $l$-chain-coterie as communication structure.

**Proof:** Process $p_i$ can directly send its message to all processes in the union of quorums which $p_i$ belongs to. The message can be forwarded (propagated) by those processes in the quorums. Because of the intersection property of $l$-chain-coterie and that the maximum length of a chain between a quorum containing $p_i$ and a quorum containing $p_j$ is at most $l - 1$, the message will arrive to $p_j$ by at most $l$ rounds of sending and receiving messages. □

**Definition 2 (Communication set)**

Process $p_i$'s *communication set* denoted by $C_i$ is the union of quorums, which $p_i$ belongs to, excluding $p_i$ itself. Every process sends/receives directly messages only to/from processes in its *communication set*. □

**A distributed method for achieving multi-synchronization (DMMS):** To maintain the synchronous property, $l$ times (rounds) of sending and receiving messages are needed by using the communication sets based on a $l$-chain-coterie. Here round $j$ of message exchanges means sending messages at Step $j$ and receiving messages at Step $j + 1$.

Every process $p_i$ executes the following $l' = l + 1$ steps.

**Step 1**

When $p_i$ becomes idle

send every process in $C_i$ a *request* message carrying $p_i$'s synchronous conditions on the event

**Step 2**

On receiving a *request* message from every process of $C_i$

let $S$ be the set containing the synchronous conditions carried by the received messages and $p_i$'s own condition;

check whether every pair of elements in $S$ are satisfied as described, in Table 1;

**IF** all synchronous conditions are satisfied

**THEN** send a *matched*(matched-condition) message to every process in $C_i$

**ELSE** send an *unmatched* message to every process in $C_i$

**Step *j*** $(3 \leq j \leq l')$

On receiving a *matched*(matched-condition) or an *unmatched* message from every process of $C_i$

**IF** all the received messages are *matched* and all the *matched-condition*s carried by the messages are satisfied with each other.

**THEN IF** $j \neq l'$

**THEN** send a *matched*(matched-condition) message to every process in $C_i$

**ELSE** execute the event; exit

**ELSE IF** $j \neq l'$

**THEN** send an *unmatched* message to every process in $C_i$

**ELSE** exit

**Lemma 2** Using *DMMS*, all processes in a process-set $P_k$ will know whether their local synchronous conditions are matched or not, by $l$ rounds of message exchange, with the mes-

sage complexity being $O(\sum_{i=1}^{n} |\cup_{p_i \in s_j} s_j|)$, if a $l$-chain-coterie is available to $P_k$.

**Proof:** We first discuss the case where the synchronous conditions on $e_k$ are not satisfied by all of the processes in a process-set $P_k$, in which a $l$-chain-coterie is employed as communication structure. In other words, there are some processes which hold the different local values or different types of variables. Without loss of the generality, let $p_i$'s local synchronous conditions be different from that of $p_j$. We show every process will know the $e_k$ is disabled by $l$ rounds of message exchanges.

A message sent by $p_i$ carrying $p_i$'s local synchronous conditions will be received by every process $p_h$ in $p_i$'s communication set, within one round of message exchanges (i.e., sent at Step 1 and received at Step 2). Process $p_h$ checks whether $p_i$'s synchronous condition is matched with $p_h$'s own condition and other conditions received from other processes, and sends a *matched* message carrying the matched conditions (e.g., the matched-value) or an *unmatched* message to every process in $p_h$'s communication set, depending on whether the conditions are matched or not. Similar to $p_h$, a process will check and send out a *matched* message carrying the matched synchronous conditions or an *unmatched* message, on receiving *matched* or *unmatched* messages from all processes in its communication set.

At step $l'$ (round $l$), $p_j$ receives either an *unmatched* message or *matched* messages carrying matched conditions which have been matched with $p_i$'s local synchronous conditions. Therefore $p_j$ will know the synchronous conditions on $e_k$ are not matched, since $p_j$'s local conditions doesn't match $p_i$'s conditions. In the same way, $p_i$ will know the synchronous conditions on $e_k$ are not matched.

Moreover, for every process $p$ in the process-set, it will receive either an *unmatched* message or *matched* messages including $p_j$'s and $p_i$'s synchronous conditions. That is, $p$ will know that the synchronous conditions are not matched.

In the case that the synchronous conditions on $e_k$ are satisfied by all of the processes in a process-set $P_k$. In other words, there are no process which holds the different local values or different types of variables. From the step 3 through step $l'$ (round 2 through $l$), all received messages are *matched* type and conditions carried by all received messages are matched with each other, so that every process will know the $e_k$ is enabled by $l$ rounds of message exchanges.

Since every process sends a message to all other processes in its quorums, the message complexity to achieve the consensus on synchronous conditions is $O(\sum_{i=1}^{n} |\cup_{p_i \in s_j} s_j|)$. □

#### 3.1.2 A Method for Creating a $l$-chain-coterie ($MCL$)

In order to show that a $l$-chain-coterie can be indeed created for a process-set and the message complexity is really improved comparing with previous researches based on $l$-chain-coterie, we present a method for creating a $l$-chain-coterie.

Similar to the coterie-based approach [9),16)], it is desired that all processes to send and receive the same number of messages in a round, to balance the load and responsibility. In other words, symmetric $l$-chain-coterie is desired.

**Definition 3** A $LCC$ is said to be symmetric, if both the following (1) and (2) are satisfied.

(1) The number of processes of every quorum is the same.

(2) The number of quorums, to which every process belongs, is the same. □

From Lemma 2, $O(n|s_i|)$ messages are required for achieving the consensus on synchronous conditions of a process-set which use a symmetric $l$-chain-coterie as a communication structure, where $s_i$ is a quorum in $LCC$.

We give the following method, simply called $MCL$, to create a symmetric $l$-chain-coterie for a process-set with $n$ processes, in the following two cases.

(1) The case of $n = \binom{m}{l}$, where $m$ is an integer.

S1: Create a $m$-vertices $l$-complete hypergraph[☆] as an auxiliary graph $AG(V, H)$, with $|V| = m$ vertices and $|H| = n = \binom{m}{l}$ hyperedges.

Here, by $l$-complete hypergraph, we mean there is a hyperedge among every $l$ vertices of the hypergraph. In other words, it is a $l$-uniform and $\binom{m-1}{l-1}$ regular hypergraph. Notice that we use the word "vertices", to represent the nodes in the auxiliary graph to differ from "processes" in the process-set.

S2: Associate each process in the process-set

[☆] A hypergraph is a pair $(V, H)$, where $V$ is a finite set of vertices and $H$ is a set of hyperedges which are arbitrary nonempty subsets of $V$. A hypergraph in which all vertices have the same degree is said to be regular. A hypergraph in which all hyperedges contain the same number of vertices is said to be uniform.

with a hyperedge of the hypergraph.
Since a process $p_i$ corresponds to a hyperedge $h_i$ that can be represented with its $l$ endpoints (vertices) in the auxiliary hypergraph, we represent $p_i$ with the set of the endpoints $v(p_i) = \{v_i^1, v_i^2, \cdots, v_i^l\}$, in some cases, for the convenience of the discussion.

S3: For every $l-1$ vertices in $AG$, a quorum $q_i$ is generated by collecting all processes whose associated hyperedges share the $l-1$ vertices as endpoints.
The set of $l-1$ vertices, denoted with $v(q_i)$, is called the core of $q_i$.

(2) The case of $n \neq \binom{m}{l}$.

The case could be dealt with by adding some virtual processes, similar to the method in Lakshman's paper [9]. Specifically, let $\binom{m-1}{l} < n < \binom{m}{l} = n'$. In addition to $n$ real processes in the process-set, $n' - n$ virtual processes are added, to construct a symmetric $l$-chain-coterie with $n'$ processes, by using the method in Case (1).

The virtual processes behave almost same as the real processes, i.e., performing the same algorithm proposed in the paper. Except that the synchronous conditions of a virtual process are dummy data which will match with any synchronous conditions of another virtual or real process.

The method of adding virtual processes can simply create a symmetric $l$-chain-coterie, though the virtual processes send dummy data which is a waste of network resources. Another possible approach is create near symmetric $l$-chain-coterie by using much more complex methods than adding virtual processes.

It is obvious that the number of quorums generated by above method is $\binom{m}{l-1}$. And the number of processes in a quorum is $(m-l+1)$.

We show some properties of the created quorums and a symmetric $l$-chain-coterie is formed by the quorums, in the rest part of the subsection.

**Definition 4 (Distance between two quorums)**

The distance between $q_i$ and $q_j$ is defined as $D(q_i, q_j) = l - 1 - |v(q_i) \cap v(q_j)|$. □

From the definition, $0 \leq D(q_i, q_j) \leq l-1$. Intuitively, the cores of $q_i$ and $q_j$ share $l - 1 - D(q_i, q_j)$ vertices in $AG(V, H)$ and one has $D(q_i, q_j)$ different vertices from another. If and only if $D(q_i, q_j) = 0$, the two quorums are identical.

**Lemma 3** When $D(q_i, q_j) = 1$, (1) $q_i$ and $q_j$ have at least one process in common, and (2) each of $q_i$ and $q_j$ has at least one process which is not contained by another.

**Proof:** Without loss of generality, we assume that $v(q_i) = \{v_1, v_2, \cdots, v_{l-1}\}$ and $v(q_j) = \{v_2, v_3, \cdots, v_l\}$, which satisfy $D(q_i, q_j) = 1$.

(1) At least the process $\{v_1, v_2, \cdots, v_{l-1}, v_l\}$, is a process shared by both quorums $q_i$ and $q_j$.

**Note:** In this paper, we say a process $p$ is shared by two or more sets to mean that $p$ belongs to intersection of the sets. Moreover, a process $p$ belongs to a set $q$, iff $p \in q$.

(2) Because of $m > l$, there is at least one vertex, say $v_{l+1}$ in $AG$, besides vertices $v_1, v_2, \cdots, v_l$. Process $\{v_1, v_2, \cdots, v_{l-1}, v_{l+1}\}$ belongs to $q_i$ but not to $q_j$. On the other hand, process $\{v_2, v_3, \cdots, v_l, v_{l+1}\}$ belongs to $q_j$ but not to $q_i$. □

**Lemma 4** For an arbitrary pair $q_1$ and $q_k$ of quorums $(1 < k \leq l)$, such that $D(q_1, q_k) = k - 1$, there exist quorums $q_i$ $(1 < i < k)$, such that $q_1 \cap q_2 \neq \phi, \cdots, q_{i-1} \cap q_i \neq \phi, \cdots, q_{k-1} \cap q_k \neq \phi$.

**Proof:** $|q_1 \cap q_k| = l - k$ due to $D(q_1, q_k) = k - 1$. Without loss of generality, let
$v(q_1) = \{v_1^1, v_1^2, \cdots, v_1^{k-1}, v^k, \cdots, v^{l-1}\}$ and
$v(q_k) = \{v_k^1, v_k^2, \cdots, v_k^{k-1}, v^k, \cdots, v^{l-1}\}$.
Here, a subscription of a vertex indicates the identifier of the quorum. The subscriptions of the vertices which are shared by both $q_1$ and $q_k$ are omitted.
There must exist the following $k-2$ different quorums:
$v(q_i) = \{v_i^1, v_i^2, \cdots, v_i^{k-1}, v^k, \cdots, v^{l-1}\}$, $\cdots$,
where $2 \leq i \leq k-1$, such that
$\forall v_i^j,\ i \in \{2, \cdots, k-1\}, j \in \{1, \cdots, k-1\}$, $v_i^j = v_k^j\ for\ i > j$, $v_i^j = v_1^j\ for\ i \leq j$.
Because $D(q_{i-1}, q_i) = 1$, $q_{i-1}$ and $q_i$ have a process in common, by Lemma 3 (1). That is, $q_{j-1} \cap q_j \neq \phi$. □

**Example 1** Let's consider the case of $l = 5$, $k = 3$, and $m \geq l + 1$. Let $v(q_1) = \{a, b, x, y\}$ and $v(q_3) = \{c, d, x, y\}$. Then there must be another quorum whose core is $v(q_2) = \{c, b, x, y\}$, such as $D(q_1, q_2) = 1$ and $D(q_2, q_3) = 1$.

**Proposition 1** A symmetric $l$-chain-coterie is formed by the quorums created the method $MCL$.

(1) Intersection property is satisfied from Lemma 4.

(2) Minimality property is satisfied from Lemma 3 (2).

Furthermore the $l$-chain-coterie is symmetric, since it is created based on a uniform and reg-

ular hypergraph. □

**Lemma 5** The number of processes in $p_i$'s communication set is $l(m-l)$.

**Proof:** A process $p_i$ belongs to $\binom{l}{l-1}$ quorums. The number of processes, including $p_i$, in every quorum is $m-l+1$. So the number of processes in the union of all the quorums excluding $p_i$ is $\binom{l}{l-1} \times (m-l)$. □

**Example 2** Given a process-set with 20 processes, i.e., $P = \{p_1, p_2, \cdots, p_{20}\}$. We use our method to create a symmetric *3*-chain-coterie.

Firstly, a *6*-vertices *3*-complete auxiliary hypergraph AG(V,H) is created, where the vertices set $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and the hyperedge set $H = \{h_1, h_2, \cdots, h_{20}\}$ with
$h_1 = \{v_1, v_2, v_3\}$, $h_2 = \{v_1, v_2, v_4\}$,
$h_3 = \{v_1, v_2, v_5\}$, $h_4 = \{v_1, v_2, v_6\}$,
$h_5 = \{v_1, v_3, v_4\}$, $h_6 = \{v_1, v_3, v_5\}$,
$h_7 = \{v_1, v_3, v_6\}$, $h_8 = \{v_1, v_4, v_5\}$,
$h_9 = \{v_1, v_4, v_6\}$, $h_{10} = \{v_1, v_5, v_6\}$,
$h_{11} = \{v_2, v_3, v_4\}$, $h_{12} = \{v_2, v_3, v_5\}$,
$h_{13} = \{v_2, v_3, v_6\}$, $h_{14} = \{v_2, v_4, v_5\}$,
$h_{15} = \{v_2, v_4, v_6\}$, $h_{16} = \{v_2, v_5, v_6\}$,
$h_{17} = \{v_3, v_4, v_5\}$, $h_{18} = \{v_3, v_4, v_6\}$,
$h_{19} = \{v_3, v_5, v_6\}$, $h_{20} = \{v_4, v_5, v_6\}$.

Secondly, let $p_i$ in process-set be associated with the hyperedge $h_i$. Finally, we have the cores for creating the quorums as follows.
$v(q_1) = \{v_1, v_2\}$, $v(q_2) = \{v_1, v_3\}$, $v(q_3) = \{v_1, v_4\}$, $v(q_4) = \{v_1, v_5\}$, $v(q_5) = \{v_1, v_6\}$, $v(q_6) = \{v_2, v_3\}$, $v(q_7) = \{v_2, v_4\}$, $v(q_8) = \{v_2, v_5\}$, $v(q_9) = \{v_2, v_6\}$, $v(q_{10}) = \{v_3, v_4\}$, $v(q_{11}) = \{v_3, v_5\}$, $v(q_{12}) = \{v_3, v_6\}$, $v(q_{13}) = \{v_4, v_5\}$, $v(q_{14}) = \{v_4, v_6\}$, $v(q_{15}) = \{v_5, v_6\}$.

Quorums created based on above cores are shown below. $q_1 = \{p_1, p_2, p_3, p_4\}$, $q_2 = \{p_1, p_5, p_6, p_7\}$,
$q_3 = \{p_2, p_5, p_8, p_9\}$, $q_4 = \{p_3, p_6, p_8, p_{10}\}$,
$q_5 = \{p_4, p_7, p_9, p_{10}\}$, $q_6 = \{p_1, p_{11}, p_{12}, p_{13}\}$,
$q_7 = \{p_2, p_{11}, p_{14}, p_{15}\}$, $q_8 = \{p_3, p_{12}, p_{14}, p_{16}\}$,
$q_9 = \{p_4, p_{13}, p_{15}, p_{16}\}$, $q_{10} = \{p_5, p_{11}, p_{17}, p_{18}\}$,
$q_{11} = \{p_6, p_{12}, p_{17}, p_{19}\}$, $q_{12} = \{p_7, p_{13}, p_{18}, p_{19}\}$,
$q_{13} = \{p_8, p_{14}, p_{17}, p_{20}\}$, $q_{14} = \{p_9, p_{15}, p_{18}, p_{20}\}$,
$q_{15} = \{p_{10}, p_{16}, p_{19}, p_{20}\}$.

The communication sets of processes are:
$C_1 = \{p_2, p_3, p_4, p_5, p_6, p_7, p_{11}, p_{12}, p_{13}\}$,
$C_2 = \{p_1, p_3, p_4, p_5, p_8, p_9, p_{11}, p_{14}, p_{15}\}$,
$C_3 = \{p_1, p_2, p_4, p_6, p_8, p_{10}, p_{12}, p_{14}, p_{16}\}$,
$C_4 = \{p_1, p_2, p_3, p_7, p_9, p_{10}, p_{13}, p_{15}, p_{16}\}$,
$C_5 = \{p_1, p_2, p_6, p_7, p_8, p_9, p_{11}, p_{17}, p_{18}\}$,
$C_6 = \{p_1, p_3, p_5, p_7, p_8, p_{10}, p_{12}, p_{17}, p_{19}\}$,
$C_7 = \{p_1, p_4, p_5, p_6, p_9, p_{10}, p_{13}, p_{18}, p_{19}\}$,
$C_8 = \{p_2, p_3, p_5, p_6, p_9, p_{10}, p_{14}, p_{17}, p_{20}\}$,

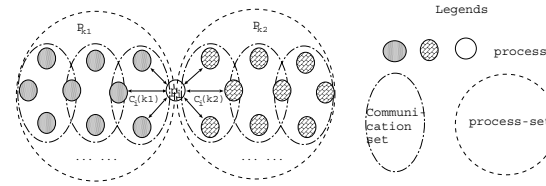


**Fig. 2** Process $p_i$ belongs to two process-sets, and sends and receives messages to/from two communication sets.

$C_9 = \{p_2, p_4, p_5, p_7, p_8, p_{10}, p_{15}, p_{18}, p_{20}\}$,
$C_{10} = \{p_3, p_4, p_6, p_7, p_8, p_9, p_{16}, p_{19}, p_{20}\}$,
$C_{11} = \{p_1, p_2, p_5, p_{12}, p_{13}, p_{14}, p_{15}, p_{17}, p_{18}\}$,
$C_{12} = \{p_1, p_3, p_6, p_{11}, p_{13}, p_{14}, p_{16}, p_{17}, p_{19}\}$,
$C_{13} = \{p_1, p_4, p_7, p_{11}, p_{12}, p_{15}, p_{16}, p_{18}, p_{19}\}$,
$C_{14} = \{p_2, p_3, p_8, p_{11}, p_{12}, p_{15}, p_{16}, p_{17}, p_{20}\}$,
$C_{15} = \{p_2, p_4, p_9, p_{11}, p_{13}, p_{14}, p_{16}, p_{18}, p_{20}\}$,
$C_{16} = \{p_3, p_4, p_{10}, p_{12}, p_{13}, p_{14}, p_{15}, p_{19}, p_{20}\}$,
$C_{17} = \{p_5, p_6, p_8, p_{11}, p_{12}, p_{14}, p_{18}, p_{19}, p_{20}\}$,
$C_{18} = \{p_5, p_7, p_9, p_{11}, p_{13}, p_{15}, p_{17}, p_{19}, p_{20}\}$,
$C_{19} = \{p_6, p_7, p_{10}, p_{12}, p_{13}, p_{16}, p_{17}, p_{18}, p_{20}\}$,
$C_{20} = \{p_8, p_9, p_{10}, p_{14}, p_{15}, p_{16}, p_{17}, p_{18}, p_{19}\}$

### 3.2 Basic Ideas of the Algorithm

Our basic idea is to employ a symmetric $l$-chain-coterie discussed in Section 3.1.2 as a communication structure for every process-set. That is, $p_i$ holds a communication set $C_i(k) \subset P_k$ for every $e_k \in E_i$. **Figure 2** shows a process $p_i$ may participate in one of two events $e_{k1}$ and $e_{k2}$. In other words, $p_i$ is shared by two process-sets $P_{k1}$ and $P_{k2}$. Therefore, $p_i$ has two communication sets $C_i(k1)$ and $C_i(k2)$. $p_i$ sends/receives messages to/from those communication sets directly.

In Step 1 through $l'$, every process-set does the same as described in *DMMS* (Section 3.1.1) in order to guarantee the synchronous property. At Step $l'$, a process $p_i$ may receive *matched* messages from all processes of more than one communication set, which means more than one event in $E_i$ are enabled.

In such a case, to maintain the exclusive property, a process shared by more than one process-set has to select only one event (process-set) to participate in. The selection has to be fair and makes progress.

We use a *counter-based* method to preserve the exclusion, fairness, and progress properties. (See Section 4, for the proof that these properties are preserved.) For every event $e_k$ in process $p_i$'s event-set $E_i$, a counter $cn_i(k)$ is used. Initially, the value of every counter is set to zero, and it will be increased during the ex-

ecution of the algorithm.

**Selection strategy:** Let $cn_i(k)$ denote $p_i$'s counter for event $e_k \in E_i$, $id(k)$ be the unique identifier of $e_k$. We assume that $id(k)$ is given to $e_k$ and a total order is defined on all $id(k)$'s in advance. We call the pair of $(cn_i(k), id(k))$ the extended identifier of $e_k$.

Process $p_i$ selects the event having the minimum $(cn_i(k), id(k))$, where

$(cn_i(k), id(k)) < (cn_i(j), id(j))$ iff
$cn_i(k) < cn_i(j)$ or
$cn_i(k) = cn_i(j)$ and $id(k) < id(j)$

Moreover, a process which participates in only one event selects the event. □

Event $e_k$ can be executed, only when all processes in $P_k$ have selected it. Therefore, a process $p_i$ needs to let other processes in process-sets sharing the process know its selection, after it selects an event (process-set) locally. To this end, $p_i$ sends a *selected* message to the communication set of the selected event. When a process receives a *selected* message from every process in a communication set, it will send a *do* message to every process in the set. If all processes in a process-set select the corresponding event and sends *selected* messages, every one will receive a *do* message, which means the event is executed, by Step $2l'$.

In order to guarantee that conflicting events can't be executed simultaneously, on receiving a *do* message, $p_i$ executes the event and sends an *unselected* message to all processes in its communication set, except the executed one.

When a process receives an *unselected* message from a process in a communication set, it will send an *undo* message to every process in the set. If some process in a process-set sends an *unselected* message, every process will receive an *undo* message, which means the event can't be executed, by Step $2l'$.

In order to preserve the fairness property, the extended identifier of the executed event is required to be changed dynamically to larger than all its conflicting events. The identifier $id(k)$ of event $e_k$ is given in advance and unchanged. However, $cn_i(k)$ is changed as follows.

A process $p_i$ sends the maximum extended identifier among all events in $E_i$, i.e., $\max_{e_l \in E_i}((cn_i(l), id(l))$, carried by the *selected* messages, to all processes in $C_i(k)$ in Step $l'+1$. On receiving a *selected(m)* from every process in $C_j(k)$, $p_j$ sends a *do(m)* message to every process in $C_j(k)$ in Steps $l'+2$, where the parameter $m$ is the maximum extended identifier among $p_j$'s own maximum extended identifier and the identifiers received in *selected* messages. Similarly, from Step $l'+3$ through Step $2l'-1$, every $p_j$ forwards a *do(m)* message to every process in $C_j(k)$, after taking $m$ to be the maximum one among its own maximum and the identifiers received in *do(m)* messages in last step, if all received messages are *do(m)* messages. In the Step $2l'$, every process $p_i$ executes the event $e_k$ and resets $cn_i(k)$ to max +1, where *max* is the maximum among $p_i$'s own maximum and the identifiers received.

### 3.3 Formal Description of the Algorithm

A process $p_i$ holds the following variables.

$E_i$ : The event-set which $p_i$ participates in.
$C_i(k)$ : The communication set for event $e_k \in E_i$. Here, $C_i(k) \subset P_k$. $P_k$ is a process-set which participate in $e_k$.
$cn_i(k)$ : The counter for event $e_k \in E_i$.
$id(k)$ : An integer indicating the identifier of $e_k \in E_i$.

Every process $p_i$ executes the following steps.

**Step 1**

When $p_i$ becomes idle
for $\forall e_k \in E_i$, send a *request* message, carrying parameters for $p_i$'s synchronous conditions on $e_k$, to every $p_j \in C_i(k)$.

**Step 2**

For $\forall e_k \in E_i$, on receiving a *request* message from every $p_j \in C_i(k)$
let $S$ be the set containing the synchronous conditions carried by the received messages and $p_i$'s own condition;
check whether every pair of elements in $S$ are satisfied (matched), as described in Table 1
**IF** all pair of conditions are satisfied
**THEN** send a *matched*(matched-condition) message to every $p_j \in C_i(k)$
**ELSE** send an *unmatched* message to every $p_j \in C_i(k)$

**Step *i*** $(3 \le i \le l')$

For every $e_k \in E_i$, on receiving a *matched*(matched-condition) or an *unmatched* message from every $p_j \in C_i(k)$
**IF** $i \neq l'$
**THEN IF** all the received messages are *matched* and all conditions carried by the received messages are satisfied with each other
**THEN** send a *matched*(matched-condition) message to every pro-

cess in $C_i$
**ELSE** send an *unmatched* message to every $p_j \in C_i(k)$
**ELSE** Let $M_i \subseteq E_i$ be the set of events, such that for every $e_k \in M_i$ the received messages from all processes in $C_i(k)$ are *matched* messages and all conditions carried by these messages are satisfied with each other
**IF** $|M_i| \geq 2$
**THEN** select only one $e_k \in M_i$ which has the smallest $(cn_i(k), id(k))$;
reject other events in $E_i - M_i$
**ELSE IF** $|M_i| = 1$
**THEN** select the event in $M_i$, and reject other events in $E_i - M_i$
**ELSE** reject all $e_k \in E_i$;
go to Step 1

**Step** $l' + 1$

For the selected $e_k$, send a *selected(m)* message to every process $p_j \in C_i(k)$, where $m = \max_{e_l \in E_i}((cn_i(l), id(l))$.

**Step** $l' + 2$

On receiving a *selected(m)* message or an *unselected* message from every $p_j \in C_i(k)$
**IF** all received messages are *selected*
**THEN** Let $m'$ denote the maximum extended identifier among $p_i$'s own maximum extended identifier and the identifiers received in *selected(m)* messages;
send a *do(m′)* to every $p_j \in C_i(k)$
**ELSE** send an *undo* to every $p_j \in C_i(k)$

**Step** ***i*** $(l' + 3 \leq i \leq 2l')$

On receiving a *do(m)* message or an *undo* message from every $p_j \in C_i(k)$
**IF** $i \neq 2l'$
**THEN IF** all the received messages are *do(m)*
**THEN** let $m'$ be the maximum among $p_i$'s own maximum extended identifier and the identifiers received in *do(m)* messages;
send a *do(m′)* to every process in $C_i$
**ELSE** send an *undo* to every $p_j \in C_i(k)$
**ELSE IF** all received messages from $C_i(k)$ are *do(m)* messages
**THEN** let max be the maximum among $p_i$'s own maximum extended identifier and the identifiers received in *do(m)* messages.
$cn_i(k) = \max + 1$;
For $\forall e_{k'} \in M_i$, *except* $e_k$, reject $e_{k'}$ and send an *unselected* message to every process $p_j \in C_i(k')$;
$state_i = active$;
execute $e_k$;
goto Step 1
**ELSE** (i.e., some received message from $C_i(k)$ is *undo*)
$M_i = M_i - \{e_k\}$;
**IF** $|M_i| \geq 2$
**THEN** select another event $e_k \in M_i$, which has the smallest $(cn_i(k), id(k))$;
go to step $l' + 1$;
**ELSE IF** $|M_i| = 1$
**THEN** select the event;
go to step $l' + 1$;
**ELSE** goto Step 1

## 4. Properties and Complexity of the Algorithm

### 4.1 Properties

We now outline a proof that the algorithm is correct. That is, we show that the algorithm maintains the properties of a Multi-Rendezvous described in Section 2.

**Lemma 6 (Synchronous property)**
The algorithm preserves that a disabled event will not be executed by the algorithm.

**Proof:** Suppose an event $e_k$ is disabled, we show the process-set $P_k$ can't execute $e_k$. Obviously if some process in $P_k$ has not become idle, $P_k$ can't execute the corresponding event. Suppose all processes in $P_k$ become idle, but the synchronous conditions on $e_k$ are not matched by all of the processes in $P_k$. From Lemma 2, all processes in $P_k$ will know their local synchronous conditions on $e_k$ are not matched, by $l$ rounds of message exchange. A process which knows the synchronous conditions on $e_k$ are not matched doesn't select $e_k$ to participate in, according to the **selection strategy**. So every $p_i$ in $P_k$ doesn't execute $e_k$. □

**Lemma 7 (Progress property)**
The algorithm preserves that if there is an enabled event, either the event or at least one of its conflicting events will be executed eventually.

**Proof:** Let's consider all enabled events and

their process-sets in a system at a time. We call two enabled events are neighboring events, if their process-sets share a common process.

Whether enabled events are selected by processes depends on their extended identifiers which are totally ordered. At least the event $e_s$ with the smallest extended identifier will be selected by all processes in its process-set. After $e_s$ is executed, we can consider it is removed, since it becomes *idle* (not longer enabled). Then there exists at least another enabled event $e_t$, such that every process of its process-set makes decision to select or reject it (see Step $l'$ and $2l'$). As a result of the decision, the process sends a *select* or an *unselect* message to other processes in the process-set. The event with the smallest extended identifier (excluding the removed $e_s$) is such an event.

There are two cases for $e_t$. If $e_t$ is a neighbor of $e_s$, the common processes shared by $e_t$'s and $e_s$'s process-sets will reject $e_t$ and send *unselected* messages to $e_t$'s process-set. Thus every process in $e_t$'s process-set will eventually receive an *undo* and $e_t$ will be removed without being executed. If $e_t$ is not a neighbor of $e_s$, it will be selected by all processes in $e_t$'s process-set, and then executed and removed.

Similarly, every enabled event $e_i$ will eventually have a chance that all processes in its process-set will decide to select or reject $e_i$ and send out a *select* or an *unselect* message to other processes in the set. i.e., $e_i$ will not be postponed forever.

If any one of $e_i$'s neighboring event is executed, an *undo* message will be received by every process in $e_i$'s process-set. Otherwise, a *do* message is received, which means $e_i$ will be executed. □

**Lemma 8 (Fairness)**

The algorithm preserves that if an event $e_k$ becomes enabled infinitely often, $e_k$ will be eventually executed, assuming that every active process will eventually become idle[☆].

**Proof:** When $e_k$'s process-set tries to participate in $e_k$, either $e_k$ or some of its conflicting event $e_l$ will be executed, as discussed in **progress** property.

If $e_l$'s extended identifier is smaller than $e_k$'s, $e_l$ will be selected and may be executed. In such a case, since the common process(es) in $P_k$ and $P_l$ participate in $e_l$, $e_k$ is rejected by these processes and can't be executed.

However, when $e_l$ is executed, $e_l$'s extended identifier will be changed to a value greater than $e_k$'s, in the Step $l'+1$ through $2l'$.

Let $|Nei|$ be the number of events which conflict with $e_k$. At most, $|Nei|$ times $e_k$ becomes enabled, the extended identifier of $e_k$ becomes the smallest one among the identifiers of its conflicting events.

When $e_k$ becomes enabled again, all processes in $P_k$ will select $e_k$ rather than its conflicting events. Thus $e_k$ will be executed eventually. □

**Lemma 9 (Exclusive property)**

The algorithm preserves that conflicting events, whose process-sets share some common process(es), will not be executed simultaneously.

**Proof:** A process $p_i$ shared by more than one process-set selects at most one event, say $s_k$ (see Step $l'$). Process $p_i$ sends *selected* messages to its communication set corresponding to $e_k$, in Step $l'+1$. If other processes in $P_k$ also select $e_k$, a *do* message will be received by $p_i$ and $e_k$ will be executed, at Step $2l'$. Then $p_i$ rejects other events conflicting with $e_k$ and sends an *unselect* message to every process $p_j$ in the communication sets corresponding to these conflicting events.

After receiving an *unselected* message, $p_j$ will send *undo* messages to its communication set (see Step $l'+2$). The *undo* messages are propagated to all processes in the unselected process-sets from Step $l'+3$ through Step $2l'-1$. Thus, all processes in a process-set will receive an *undo* message at Step $2l'$, if some process (even one process) in the process-set rejects and sends an *unselected* message.

The processes received *undo* can't execute the corresponding event. Thus, no more than one process-set execute their events simultaneously, if these process-sets share a common process. □

From the above lemmas and the algorithm, we have the following theorem.

**Theorem 1** The algorithm solves the Multi-Rendezvous Implementation Problem defined in Section 2, with the synchronous, exclusive, fairness, and progress properties being satisfied.

### 4.2 Complexity

We show message complexity of the distributed algorithm below. First, we give some notations. Let $N = |P|$ be the number of pro-

☆ Without the assumption, the fairness is impossible to be guaranteed, see Ref. 10) for the detail of the impossibility.

cesses in a network. Generally, the $N$ processes form $|E|$ process-sets which participate in $|E|$ events. Let $N_k$ be the number of processes of process-set $P_k$. A communication structure based on symmetic $l$-chain-coterie as described in Section 3.1.2 is used in $P_k$. A communication set of a process in $P_k$ has $l(m_k - l)$ processes by Lemma 5, where $m_k$ satisfies $N_k = \binom{m_k}{l}$.

The number of messages required for one event $e_k$ to be executed by process-set $P_k$ is calculated as follows.

In Step $i$, $1 \leq i \leq l$ and $l+2 \leq i \leq (2l+1)$, every process in the process-set sends a message to every process in its communication set $C_i(k)$. The number of processes in $C_i(k)$ is $l(m_k - l)$, thus the number of messages used in $P_k$ for an execution of $e_k$ is:

$$M_k = 2l \times N_k \times l(m_k - l) = 2l^2 N_k (m_k - l)$$

Because of $N_k = \binom{m_k}{l}$,

$$O(m_k) = O(N_k^{1/l})$$

Thus we obtain

$$O(M_k) = O(N_k^{(1+1/l)})$$

Besides the above messages, a process shared by $P_k$ and $P_l$ sends messages such as *unselected*, *undo* etc. to processes in $P_l$. Let $|Nei|$ be the number of events which may conflict with $e_k$ and $N_{\max}$ be the maximum number of processes of a process-set. In the worst case, the total message $M'_k$ for an execution of an event $e_k$ is

$$|Nei| \cdot 2l^2 N_{\max}(m_{\max} - l)$$

$|Nei|$ is generally a constant, Thus we have

$$O(M'_k) = O(N_{\max}^{(1+1/l)}) = O(N^{(1+1/l)})$$

## 5. Conclusions

In this paper, we first presented a definition of $l$-chain-coterie and discussed its properties, and then gave a method for creating symmetric $l$-chain-coterie based on a uniform and regular hypergraph. A new efficient distributed algorithm was developed for implementation of a Multi-Rendezvous by using the symmetric $l$-chain-coterie as communication structure. The algorithm preserves synchronous and exclusive properties of a Multi-Rendezvous. The implementation made by the algorithm is fair and makes progress.

Our algorithm requires $O(N^{1+1/l})$ messages, where $N$ is the number of the processes in the network, as opposed to $O(N^2)$ messages needed in Gao's approach [2], and $O(N^{1.5})$ in Cheng, et al.[16]. And our algorithm is fully distributed in the sense it does not use binary-tree [3] or event manager [4),5]. In addition, it does not use auxiliary resources such as tokens or forks [4),5].

Generally, an event does not need participation of all processes in its process-set. It can be participated by only a subset of its process-set. How to extend our method to the general case is left as a future work.

**Acknowledgments** We would like to thank the anonymous referees for their invaluable comments, which significantly helped to improve the quality of the paper.

## References

1) ISO: Information Processing Systems-Open System Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, ISO 8807 (Feb. 1989).
2) Gao, Q. and Bochmann, G.v.: A Virtual Ring Algorithm for the Implementation of Multi-Rendezvous, Technical Report 675, Université de Montréal, Dept. IRO (1989).
3) Sisto, R., Ciminiera, L. and Valenzano, A.: A Protocol for Multi-rendezvous of LOTOS Processes, *IEEE Trans. Comput.*, Vol.40, No.1, pp.437–447 (1991).
4) Chandy, K.M. and Misra, J.: *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, MA (1988).
5) Bagrodia, R.: A Distributed Algorithm to Implement N-Party Rendezvous, *LNCS*, Vol.287, pp.138–152, Springer-Verlag (1987).
6) Bagrodia, R.: Process Synchronization: Design and Performance Evaluation of Distributed Algorithms, *IEEE Trans. Softw. Eng.*, Vol.15, No.9, pp.1053–1065 (1989).
7) Maekawa, M.: A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems, *ACM Trans. Comput. Syst.*, Vol.3, No.2, pp.145–159 (1985).
8) Kumar, A.: Hierarchical Quorum Consensus: A new method for managing replicated data, *IEEE Trans. Comput.*, Vol.40, No.9, pp.996–1104 (1991).
9) Lakshman, T.V. and Agrawala, A.K.: Efficient decentralized consensus protocols, *IEEE Trans. Softw. Eng.*, Vol.SE-12, No.5, pp.600–607 (1986).
10) Tsay, Y.-K. and Bagrodia, R.L.: Some impossibility results in interprocess synchronization, *Distributed Computing*, Vol.6, pp.221–231 (1993).
11) Kakugawa, H., Fujita, S., Yamashita, M. and Ae, T.: Availability of $k$-Coterie, *IEEE Trans. Comput.*, Vol.42, No.5, pp.553–558 (1993).

12) Kuo, Y.-C. and Huang, S.-T.: A Geometric Approach for Constructing Coteries and $k$-Coteries, *IEEE Trans. Parallel and Distributed Systems*, Vol 8, No.4, pp.402–411 (1997).
13) Francez, N., Hailpern, B. and Taubenfeld, G.: Script: A communication abstraction mechanism, *Sci. Comput. Program.*, Vol.6, No.1 (1986).
14) Francez, N. and Forman, I.: *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming*, Addison Wesley, Reading, MA (1994).
15) Fujita, S.: A Quorum Based k-Mutual Exclusion by Weighted k-Quorum Systems, Information Processing Letters, Vol.67, No.4, pp.191–197 (1998).
16) Cheng, Z., Huang, T. and Shiratori, N.: A new distributed algorithm for implementation of LOTOS multi-rendezvous, *Formal Description Techniques*, VII, pp.493–504, CHAPMAN & HALL (1994).

(Received November 11, 1998)
(Accepted November 4, 1999)

**Zixue Cheng** received the M.E. and Ph.D. degrees from Tohoku University in 1990 and 1993, respectively. He was an assistant professor from 1993 to 1999 and has been an associate professor since Apr. 1999, in the Department of Computer Software at the University of Aizu. Currently he is working on distributed algorithms, network agents, and protocol synthesis and implementation. Dr. Cheng is a member of IEEE, ACM, IEICE, and IPSJ.

**Kshirasagar Naik** received his B.S., M.Tech., M.Math., and Ph.D. degrees from Sambalpur University (India), IIT Kharagpur (India), University of Waterloo (Canada), and Concordia University (Canada), respectively. From mid 1993 to mid 1999, he was on the faculty of Computer Science and Engineering at the University of Aizu, Japan. At present he is an associate professor in the School of Computer Science at Carleton University in Ottawa. His current research interests are in the area of mobile communication and computing.

**Naka Tajima** received B.S. and M.S. degrees from the University of Aizu. Currently he is working in SOGO KEIBI HOSHO CO., LTD. He is interested in distributed algorithms and is a member of IPSJ.

**Tongjun Huang** received M.E. degree from the University of YanShan in China. Since 1993, she has been a research associate in the Information Systems and Technology Center at the University of Aizu. She is interested in CSCW, groupware, and distributed algorithms.

**Shoichi Noguchi** received his B.E., M.E., and D.E. degree in Electrical Communication Engineering from Tohoku University. He joined the Research Institute of Electrical Communication at Tohoku University in 1960, since 1971 being a professor at the same university. Dr. Noguchi was Director of Research Center for Applied Information Science, Tohoku University, from 1990 to 1993. He was a professor in Japan University from 1993 to 1997, and has been the president of the University of Aizu, since Apr. 1997. His main area of interests is information science theory and computer network fundamentals. He has also active interest in the areas of parallel processing, computer network architecture, and knowledge engineering fundamentals.