Coordinating First-Order Multiparty Interactions

YUH-JZER JOUNG
National Taiwan University
and
SCOTT A. SMOLKA
SUNY at Stony Brook

A *first-order multiparty interaction* is an abstraction mechanism that defines communication among a set of *formal process roles*. Actual processes participate in a first-order interaction by *enroling* into roles, and execution of the interaction can proceed when all roles are filled by distinct processes. As in CSP, enrolement statements can serve as guards in alternative commands. The *enrolement guard scheduling problem* then is to enable the execution of first-order interactions through the judicious scheduling of roles to processes currently ready to execute enrolement guards.

We present a fully distributed and message-efficient algorithm for the enrolement guard scheduling problem, the first such solution of which we are aware. We also describe several extensions of the algorithm, including *generic roles*, *dynamically changing environments* where processes can be created and destroyed at run time, and *nested-enrolement* which allows interactions to be nested.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management—*synchronization*; D.4.4 [**Operating Systems**]: Communications Management—*input/output*; *message sending*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General terms: Algorithms, Design, Languages

Additional Key Words and Phrases: Multiparty interaction, first-order interaction, distributed languages, distributed algorithms, interaction scheduling, IP, rendezvous, committee coordination

1. INTRODUCTION

A *multiparty interaction* is a set of I/O actions executed jointly by a number of processes, each of which must be ready to execute its own action for any of the actions in the set to occur. An attempt to participate in an interaction delays a process until all other participants are available. After the actions are executed, the participating processes continue their local computation, usually asynchronously. Languages like CSP [Hoare 1978] and Ada [U.S. Department of Defense 1983] support interaction only between two processes. However, for many applications a higher level of ab-

This research was supported, in part, by NSF Grants CCR–8704309, CCR–9120995 and CCR–9208585, and AFOSR Grant F49620-93-1-0250DEF.
Authors' addresses: Yuh-Jzer Joung, Department of Information Management, National Taiwan University, Taipei, Taiwan; Scott A. Smolka, Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794–4400, U.S.A.

straction can be obtained by permitting interaction among an *arbitrary* number of processes. For example, consider the well-known Dining Philosophers problem [Dijkstra 1965]. The natural unit of process interaction in this setting is between a philosopher and its two neighboring forks; i.e., a multiparty synchronization involving three processes.

Zeroth-Order vs. First-Order Multiparty Interaction

It is useful to distinguish between *zeroth-order* multiparty interactions where the participants are fixed in advance, and *first-order* multiparty interactions where the participants may vary dynamically. For example, a zeroth-order *multicast* interaction in which process P sends a message m to Q and R is captured by the code:

$$
\begin{aligned}
&P :: b[\,]\\
&Q :: b[x := m]\\
&R :: b[y := m]
\end{aligned}
$$

The notation we use is based on IP [Francez and Forman 1990b; Francez and Forman 1994]: b is the interaction name and m, x, and y are variables local to P, Q, and R, respectively. An interaction body $[\ldots]$ describes the action taken by a process engaged in the interaction; e.g., $x := m$ means that Q assigns variable x the non-local value of m. Many of the existing constructs for multiparty interaction are zeroth-order, including *shared actions* [Ramesh and Mehndiratta 1985], *joint actions* [Back and Kurki-Suonio 1988], *interactions* in Raddle [Forman 1986], *interactions* in IP, and *interaction types* [Kumar 1990].

A first-order multiparty interaction is an abstraction mechanism that defines activities among a set of *roles*, which serve as formal process parameters. Actual processes participate in an interaction by enroling[1] into the roles. When all roles are filled by distinct processes, an instance of the interaction is activated and the enrolers then communicate in the manner as described by the interaction.

Consider once again the multicast example. We can define a first-order interaction having three roles, a transmitter and two recipients, as follows:

$$
\begin{aligned}
&\textbf{interaction}\ \textit{bi-cast} ::\\
&[\quad \textbf{role}\ \textit{transmitter}(\textbf{value}\ z\text{: }\textbf{integer}) :: \textbf{skip}\\
&\quad\ \ \textbf{role}\ \textit{recipient}_1(\textbf{result}\ x\text{: }\textbf{integer}) :: \quad x := z\\
&\quad\ \ \textbf{role}\ \textit{recipient}_2(\textbf{result}\ y\text{: }\textbf{integer}) :: \quad y := z \quad]
\end{aligned}
$$

Now, processes P, Q, and R can execute the following enrolement statements to activate an instance of the *bi-cast* interaction:

$$
\begin{aligned}
&P :: \textbf{enrole}\ \textit{transmitter}(m)@\textit{bi-cast}\\
&Q :: \textbf{enrole}\ \textit{recipient}_1(x)@\textit{bi-cast}\\
&R :: \textbf{enrole}\ \textit{recipient}_2(y)@\textit{bi-cast}
\end{aligned}
$$

The concept of first-order interaction first appeared in the literature in a bi-party form. For example, the *rendezvous* in Ada is an asymmetric first-order interaction involving only two roles, one assumed by a fixed callee (server task) and the other by various callers (client tasks). Symmetric bi-party interaction is supported in

[1] N. Francez and I. R. Forman suggest the new word *enrole* to mean "to enter a role."

CCS [Milner 1989], where an interaction a is realized through the synchronous execution of complementary actions a and $\overline{a}$. A process ready to perform a can establish an interaction with any process ready to perform $\overline{a}$, and vice versa.

Support for first-order interactions of arbitrary arity can be found in a number of distributed programming languages. The first-order interaction considered in this paper is similar to the *script* mechanism [Francez, Hailpern, and Taubenfeld 1986]. The *compact* construct [Charlesworth 1987] generalizes Ada's bi-party rendezvous to *multiway* involving a callee and an arbitrary number of callers. As in Ada, the role of the callee is assumed by a fixed process while the caller role can be assumed by different processes.

Although the basic interaction construct of IP and its predecessor Raddle is zeroth-order, first-order mutliparty interactions can be achieved through the use of *teams*. Recall that interaction statements in IP have the form $a[\ldots]$, where a is the interaction name and $[\ldots]$ describes the interaction body. A process P is a participant of interaction a if such an expression appears in P's code, where P can be an actual executing agent or a formal parameter (role). Formal processes are declared within teams, which serve as the encapsulation and modularization mechanism in IP and Raddle. For example, the first-order broadcast example discussed above corresponds to interaction a in the following team:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{team}\ \textit{bi-cast} ::} \\
[& \textbf{role}\ \textit{transmitter}(\textbf{value}\ z\text{: }\textbf{integer}) :: a[\ ] \\
\| & \textbf{role}\ \textit{recipient}_1(\textbf{result}\ x\text{: }\textbf{integer}) :: \ a[x := z] \\
\| & \textbf{role}\ \textit{recipient}_2(\textbf{result}\ y\text{: }\textbf{integer}) :: \ a[y := z] \\
] &
\end{array}
$$

Note that teams may contain actual processes as well. So an interaction may involve formal roles and actual processes. For example, the interaction *produce* in the following team [Francez and Forman 1990b] has two participants, role *sender* and process *buffer*:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{team}\ \textit{Bounded-Buffer}\ (\textbf{value}\ \textit{bound}\text{: }\textbf{integer}) ::} \\
[& \textbf{role}\ \textit{sender}(\textbf{value}\ x\text{: }\textbf{integer}) :: \textit{produce}[\ ] \\
\| & \textbf{role}\ \textit{receiver}(\textbf{result}\ y\text{: }\textbf{integer}) :: \textit{consume}[y := \textit{first}(q)] \\
\| & \textbf{process}\ \textit{buffer}:: \\
 & \qquad \textit{size} : \textbf{integer} := 0;\ \ q : \textbf{queue} := \textit{nil}; \\
 & \qquad *[\ \ \textit{size} < \textit{bound}\ \&\ \textit{produce}[\ q := \textit{enqueue}(x, q)\ ] \longrightarrow \textit{size} := \textit{size} + 1 \\
 & \qquad \Box\ \ \textit{size} > 0\ \&\ \textit{consume}[\ q := \textit{dequeue}(q)\ ] \longrightarrow \textit{size} := \textit{size} - 1\ ] \\
] &
\end{array}
$$

Similarly, interaction *consume* has two participants, role *receiver* and process *buffer*. Note that guards in IP, like in CSP, may have a Boolean component in addition to an interaction component. Both parts are optional.

In [Joung and Smolka 1992], we present a taxonomy of programming languages offering linguistic support for multiparty interaction, along with a comprehensive complexity analysis of the multiparty interaction implementation problem.

Concurrent Activations

Since any set of processes can use/reuse a first-order interaction to communicate, a decision must be made whether to allow concurrent activations of the same first-order interaction. In Raddle and scripts, first-order interactions are reentrant thereby permitting concurrent activations, while in IP, this decision is currently left unspecified. Here we also view first-order interactions as reentrant: the potential concurrency can lead to increased performance, and, as pointed out by Francez [1990], the absence of any sequentialization of activations leaves the interaction itself as the sole source of process synchronization. However, as also pointed out in [Francez 1990], the accompanying proof system may become more complex in light of concurrent activations.

Note that concurrent activations in Ada and compacts are *not* possible because each interaction involves a role that can only be assumed by a fixed process (the callee).

Interaction Guards

As in CSP, multiparty interactions—both zeroth-order and first-order—can serve as guards in alternative and repetitive commands. In the zeroth-order case, an interaction statement appearing as a guard can be executed only if all the other participants are in agreement. For example, consider the following program:

$$\begin{array}{lll} P :: *[\ a[\,] \longrightarrow \ldots & Q :: *[\ a[\,] \longrightarrow \ldots & R :: *[\ a[\,] \longrightarrow \ldots \\ \qquad\ \ \square\, b[\,] \longrightarrow \ldots\ ] & \qquad\ \ \square\, c[\,] \longrightarrow \ldots\ ] & \qquad\ \ \square\, c[\,] \longrightarrow \ldots\ ] \end{array}$$

Here interaction a has a fixed set of participants P, Q, and R, and so P can execute $a[\,]$ if and when Q and R agree to execute their own $a[\,]$ actions. The guard-scheduling problem for zeroth-order interactions has been elegantly characterized by Chandy and Misra [1988] as one of *Committee Coordination*: each professor in a university serves on one or more committees, the members of which are fixed; furthermore, a committee cannot convene until all its members are present. As such, no two committees with a common member can convene simultaneously.

For the first-order case, an enrolement statement appearing as a guard in a process P can be executed only if a set of processes, including P itself, agree to fill all roles of the targeted interaction.[2] To illustrate, consider the following program:

$$\begin{array}{ll} P :: *[\ \textbf{enrole}\ s_1@S \longrightarrow \ldots & Q :: *[\ \textbf{enrole}\ s_2@S \longrightarrow \ldots \\ \qquad\ \ \square\, \textbf{enrole}\ t_1@T \longrightarrow \ldots\ ] & \qquad\ \ \square\, \textbf{enrole}\ s_3@S \longrightarrow \ldots\ ] \\ R :: *[\ \textbf{enrole}\ s_2@S \longrightarrow \ldots & U :: *[\ \textbf{enrole}\ s_1@S \longrightarrow \ldots \\ \qquad\ \ \square\, \textbf{enrole}\ t_1@T \longrightarrow \ldots\ ] & \qquad\ \ \square\, \textbf{enrole}\ t_2@T \longrightarrow \ldots\ ] \end{array}$$

S and T are first-order interactions having roles s_1, s_2, s_3 and t_1, t_2, respectively. P can execute **enrole** $s_1@S$ if and when two other processes agree to enrole into s_2 and s_3. Note that both Q and R may potentially enrole into s_2.

The following issues are peculiar to the implementation of first-order guards:

[2] In IP the situation is somewhat different. Enrolement into a team does not delay a process, but rather the interactions to be executed first within the role [Francez and Forman 1990a]. We defer discussion of this point to Section 6.2.

— Since the participants of a first-order interaction may vary dynamically, determining which set of processes together can enrole in an instance of the interaction is more complex than in the zeroth-order case. In fact, we show that it is closely related to the *maximum matching* problem [Hopcroft and Karp 1973].

— Instances of a first-order interaction can be concurrently activated.

— As more than one process can enrole into the same role, the failure of any one of them should not preclude the others from filling the role. Thus, first-order interactions are inherently fault-tolerant.

The distributed algorithms of [Ramesh 1987; Chandy and Misra 1988; Bagrodia 1989a; Kumar 1990; Park and Kim 1990] for multiparty interaction guard scheduling are zeroth-order in nature. The algorithms of [Chandy and Misra 1988; Bagrodia 1989a; Park and Kim 1990] employ a fixed coordinator for each interaction and thus cannot realize concurrent activations of a first-order interaction, nor can they tolerate the failure of the coordinator. The algorithms of [Ramesh 1987; Kumar 1990] employ mutual coordination among the participants, but are highly dependent on the fact that the participants of each interaction are fixed. Therefore, there is no straightforward way to adapt these algorithms to a first-order setting.

Summary of Technical Results

The main contribution of this paper (Section 3) is a distributed algorithm for the enrolement guard scheduling problem, the first first-order solution of which we are aware. The algorithm admits the possibility for any process, upon reaching an alternative command, to behave as the coordinator of a first-order interaction. Thus, unlike the zeroth-order algorithms, it permits concurrent activations: each activation will have its own coordinator. It is also fault-tolerant as the coordinator is not designated in advance. Therefore, the failure of a potential coordinator does not preclude another potential coordinator from establishing the interaction. Moreover, the algorithm has low message complexity: at most $6m$ messages per process, per interaction guard, where m is the number of processes that can potentially enrole into the interaction.

In our algorithm, a bipartite graph data structure is used to associate processes with their potential roles, and the coordinator is required to perform *maximum matching* on this graph to decide which processes to schedule. The best-known bound for this problem is currently $O(n^{2.5})$ time, where n is the number of nodes in the graph [Hopcroft and Karp 1973]. Section 5 shows that under the restriction that a process can target at most one role of an interaction, maximum matching can be safely avoided and, instead, a coordinator can decide which processes to schedule in linear time. Moreover, through the use of *generic roles*, we argue that this restriction on alternative commands is often acceptable in practice.

Section 6 describes several extensions of our algorithm, including: *dynamically changing environments* where processes may be created and destroyed at run time, a problem suggested in [Francez, Hailpern, and Taubenfeld 1986; Francez and Forman 1990a]; interaction scheduling in IP; and *nested enrolement* where enrolement statements may appear in role bodies. Section 7 concludes.

2. A SIMPLE MODEL OF FIRST-ORDER MULTIPARTY INTERACTION

We consider a distributed system to be a finite set of concurrent processes that interact by engaging in first-order multiparty interactions. Each interaction[3] I has a fixed set $\mathcal{R}(I)$ of roles into which processes can enrole. For each role $r \in \mathcal{R}(I)$ there is a fixed set $\mathcal{P}(r)$ of processes that can potentially enrole into role r. Furthermore, we let $\mathcal{P}(I) = \bigcup_{r \in \mathcal{R}(I)} \mathcal{P}(r)$, i.e., the set of processes that can potentially enrole into some role of I. We assume that $\mathcal{R}(I)$ and each $\mathcal{P}(r)$ are finite and known to every process in $\mathcal{P}(I)$. (For each interaction I in a given first-order program, $\mathcal{R}(I)$ and each $\mathcal{P}(r)$ can be determined at compile time, provided of course there is no dynamic creation of processes.)

Initially, each process in the model is in its *local computing phase* which does not involve any interaction with other processes. Periodically, a process enters its *enrolement phase* where it is ready to enrole into any single role from a set of potential roles, possibly from different interactions. We define a *quorum* of I to be a total, 1-to-1 function from the set of roles of I to the set of processes that are ready to enrole into these roles. Note that this definition imposes three restrictions on the mapping from roles to processes, each of which is essential to the definition of a quorum: (1) the mapping is total, and thus an interaction can occur only when all roles are filled; (2) the mapping is 1-to-1, and thus the roles must be filled by distinct processes; and (3) the mapping is functional, and therefore each role is filled by at most one process. Relaxing any of these restrictions would invalidate the definition.

For example, consider an interaction I having three roles r_1, r_2 and r_3, where processes P_1 and P_2 are ready to enrole into r_1, P_1, P_3 and P_4 are ready to enrole into r_2, and P_5 and P_6 are ready to enrole into r_3:

$$I :: \begin{cases} r_1 & : P_1, P_2 \\ r_2 & : P_1, P_3, P_4 \\ r_3 & : P_5, P_6 \end{cases}$$

Then, $\{(r_1, P_1), (r_2, P_3), (r_3, P_5)\}$, $\{(r_1, P_2), (r_2, P_1), (r_3, P_6)\}$, and $\{(r_1, P_2), (r_2, P_4), (r_3, P_6)\}$ are quorums of I, but $\{(r_1, P_1), (r_1, P_2), (r_3, P_5)\}$ and $\{(r_1, P_1), (r_2, P_1), (r_3, P_5)\}$ are not.

Two quorums are *disjoint* if they are range-disjoint. We often use the more colloquial "do not involve a common process" to describe disjoint quorums. For instance, $\{(r_1, P_1), (r_2, P_3), (r_3, P_5)\}$ and $\{(r_1, P_2), (r_2, P_4), (r_3, P_6)\}$ are disjoint, but $\{(r_1, P_1), (r_2, P_3), (r_3, P_5)\}$ and $\{(r_1, P_2), (r_2, P_1), (r_3, P_6)\}$ are not because they both involve P_1.

Note that an existing quorum may cease to exist — due to a process P_i involved in the quorum returning to its local computing phase (after having enroled in an interaction) — and later exist again — due to P_i entering another enrolement phase. Thus, it is useful to distinguish between a quorum (a static entity) and an *instance of a quorum* (a dynamic entity). We will henceforth write just "quorum" to mean the longer "instance of a quorum," unless stated otherwise.

The processes involved in a quorum of an interaction I together can *establish*

[3]Henceforth, unless stated otherwise, by *interaction* we mean of the first-order type.

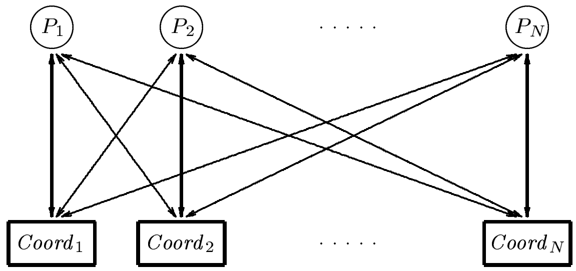

Fig. 1. The network topology.

the quorum by activating an instance of I with P_i filling role r, for each (r, P_i) in the quorum. After the interaction, the processes return to their respective local computing phases. The *enrolement guard scheduling problem* then is to devise an algorithm to establish quorums satisfying the following requirements:

Safety. At any time the established quorums are pairwise disjoint.

Liveness. If there exists a quorum, then eventually some process involved in the quorum will be placed in an established quorum.

3. AN ALGORITHM FOR ENROLEMENT GUARD SCHEDULING

In this section we present a distributed algorithm, the *EG* algorithm, for Enrolement Guard scheduling. We then prove the EG algorithm correct and analyze its message complexity.

3.1 Overview of the EG Algorithm

We assume that processes communicate exclusively by reliable asynchronous message passing. Furthermore, messages are timestamped and processes use the timestamps to maintain their logical clocks [Lamport 1978]: when a process P_i receives a message with timestamp t, P_i advances its logical clock to $\max(t, LC(P_i)) + 1$, where $LC(P_i)$ is the current value of P_i's logical clock. Initially, $LC(P_i) = 0$.

Each process P_i in the system is paired with a coordinator process $Coord_i$, which P_i "activates" when entering an enrolement phase. In our algorithm, processes communicate exclusively with coordinators and vice versa, i.e., the network topology is bipartite (see Figure 1).

Upon activation, $Coord_i$ chooses an interaction I in which P_i is ready to enrole, and attempts to *capture* processes to establish a quorum of I. $Coord_i$ builds its quorum incrementally. Initially, $Coord_i$ captures P_i and the *partial quorum* of I is

$$\{ (r, P_i) \mid P_i \text{ is ready to assume role } r \text{ of } I \}$$

$Coord_i$ then tries to capture processes to fill the other roles of I. To do so, $Coord_i$ chooses a process P_j in $\mathcal{P}(I)$ and sends it a *capture request*. If P_j is ready to assume some role of I, and is willing to be captured by $Coord_i$, P_j *grants* $Coord_i$'s capture request and provides $Coord_i$ with the set R of roles of I that P_j is ready to assume. $Coord_i$ then adds the pairs (r, P_j) for each $r \in R$ to its partial quorum. If, however, P_j *rejects* $Coord_i$'s request because it is not ready to assume a role of I or because it does not wish to be captured by $Coord_i$, then $Coord_i$ must find another process for I.

It is important to note that $Coord_i$ attempts to capture processes *one at a time*: it sends a capture request to one process and waits for a response before trying another one. $Coord_i$'s efforts succeed when its partial quorum contains a quorum. It then directs each process of the quorum to assume its role in the quorum, and *releases* any additional processes it may have captured. If, however, $Coord_i$ determines that there is a role of I for which it cannot capture a process (a point on which we elaborate in Section 3.2), it aborts the attempt and releases all members of its partial quorum *except* P_i. If P_i has another interaction I' in which it is ready to enrole, then $Coord_i$ attempts to build a quorum of I' by starting with the new partial quorum $\{ (r, P_i) \mid P_i \text{ is ready to assume role } r \text{ of } I' \}$. Otherwise, $Coord_i$ releases P_i and stops its quorum-building activity. P_i then waits to be captured by another coordinator.

The main techniques used in the EG algorithm are:

(1) Upon entering an enrolement phase, each process P_i is assigned a unique *entry time* obtained from the contents of P_i's logical clock.[4] Entry times provide a notion of relative *age* among processes: the larger the entry time a process possesses, the younger the process is. The age of a coordinator $Coord_i$ is taken to be the same as the age of P_i.

(2) *A process can be captured by only one coordinator at a time.* This ensures that coordinators establish quorums that involve pairwise disjoint sets of processes; i.e., the algorithm satisfies the safety condition for enrolement guard scheduling.

(3) Two or more coordinators seeking to establish quorums involving a common process are said to *conflict*. Specifically, a conflict arises when a coordinator attempts to capture a process P that already belongs to another coordinator.
Conflicts are resolved in favor of older coordinators in order to prevent a coordinator from being locked out from capturing processes. That is, we let the older of the two coordinators capture/retain the process P under contention. The younger coordinator has then to find an alternative process for the same role as played by P, or wait hoping that P will eventually be released by the older coordinator either because the older coordinator could not build a quorum or because it decided on a quorum in which P was not included.

(4) The techniques used to implement the previously stated policy of favoring older coordinators in a conflict are as follows. Let $Coord_i$ and $Coord_k$ be two coordinators, both of which wish to capture P_j, and assume $Coord_i$ is younger than $Coord_k$. There are two cases to consider, depending on which of the two coordinators P_j is currently captured by: (1) While captured by $Coord_k$, P_j receives a capture request from $Coord_i$. In this case, P_j sends $Coord_i$ a message of type *Deferred* . (2) While captured by $Coord_i$, P_j receives a capture request from $Coord_k$. In this case, P_j asks $Coord_i$ to release it by sending $Coord_i$ a *Switch* message. Assuming the message is not "out-of-date," $Coord_i$ will do so and acknowledge P_j's request with a *Switch_ok* message. In either case (1) or (2), we say that P_j *defers* $Coord_i$'s capture request.

[4]Entry times can be made unique by additionally considering process id's.

A coordinator $Coord_i$, upon receiving word that its capture request to P_j has been deferred, waits to hear from P_j until one of the following occurs:

—$Coord_i$ succeeds in building a quorum (which does not involve P_j) or fails in its attempt. In either case it *withdraws* its request to capture P_j.

—P_j's captor successfully builds a quorum involving P_j. So P_j rejects $Coord_i$'s capture request.

—P_j's captor releases P_j as it cannot build a quorum involving P_j. So P_j now grants $Coord_i$'s capture request.

While waiting to hear from P_j, we say that $Coord_i$'s capture request is being *maintained as deferred* by P_j. Note that during this period, $Coord_i$ does not necessarily remain idle; rather, it may continue to make progress toward its goal of establishing a quorum by attempting to capture other processes.

(5) *A coordinator is only allowed to capture older processes.* This implies that only the youngest process in a quorum can assemble the quorum, which prevents other processes from wasting messages in an attempt to build the quorum only to learn that a younger process is not yet ready. This technique thereby reduces the message complexity of the algorithm, and, as discussed below, plays a role in increasing the level of concurrency in the system. It also facilitates extending the algorithm to dynamically changing environments where processes are created and destroyed at run time (see Section 6.1).

We show in Section 4 (Corollary 4.4) that our algorithm guarantees that any quorum contained in the partial quorum of $Coord_i$ involves P_i. This prevents $Coord_i$ from coordinating one quorum after another from a given enrolement phase, assuming that $Coord_i$ is old enough to win most contentions. Allowing $Coord_i$ to coordinate too many quorums could severely limit the system's concurrency. In fact we can show that if for a given enrolement phase there exists at least k quorums that involve pairwise disjoint sets of processes, then at least k coordinators will establish quorums that involve pairwise disjoint sets of processes.

3.2 The EG Algorithm

We now present the details of the EG algorithm. First, we list the types of messages transmitted between coordinators and processes. Messages from a coordinator $Coord_i$ to a process P_j are of the following types:

—*Request* (t, I): $Coord_i$ requests P_j for capture, where t is the entry time of P_i and I is the interaction for which $Coord_i$ is attempting to establish a quorum.

—*Abort*: $Coord_i$ withdraws its capture request to P_j; $Coord_i$ also releases P_j if P_j is currently captured by $Coord_i$.

—*Success*(I, r): $Coord_i$ releases P_j and directs P_j to enrole in interaction I as role r.

—*Switch_ok*: $Coord_i$ releases P_j to resolve a conflict; $Coord_i$'s capture request is now deferred by P_j.

The messages transmitted from P_j to $Coord_i$ are of the following types:

—*Start*(*ready-set*, et_i): P_i activates $Coord_i$ and provides it with *ready-set* and et_i (see P_i's local variables below).

—*Yes*(t, I, R) : in response to $Coord_i$'s capture request *Request* (t, I), P_j grants the request and informs $Coord_i$ of the set R of roles that P_j is ready to assume.

—*No*(t, I) : in response to $Coord_i$'s capture request *Request* (t, I), P_j rejects the request.

—*Deferred* (I, R) : in response to $Coord_i$'s capture request *Request* (t, I), P_j informs $Coord_i$ that P_j is willing to assume any one of the set R of roles but the capture request is deferred by P_j.

—*Switch* : P_j requests to be released by $Coord_i$ because an older coordinator is attempting to capture P_j.

The variables local to each $Coord_i$ are as follows:

—*chosen* (I) : a flag that is true iff $Coord_i$ has chosen to coordinate the building of a quorum for interaction I.

—*active* : a flag that is set to true when P_i has entered its enrolement phase and activated $Coord_i$, and is set to false when $Coord_i$ has ceased its quorum-building activity in the enrolement phase. The initial value of *active* is *false*.

—*Int* : the interaction for which $Coord_i$ is building a quorum. The initial value of *Int* is *nil*.

—*target* : the process to which $Coord_i$ has sent a capture request and from which $Coord_i$ is waiting for a response. The initial value of *target* is *nil*.

—*p-quorum* : the partial quorum currently built by $Coord_i$. It contains pairs (r, P_j) such that P_j is captured by $Coord_i$ and is ready to assume role r. The initial value of *p-quorum* is $\emptyset$.

—*d-quorum* : set of pairs (r, P_j) such that P_j is ready to assume role r but it has deferred the capture request by $Coord_i$. The initial value of *d-quorum* is $\emptyset$.

—*requested-procs* : set of processes $Coord_i$ has requested for capture and from which $Coord_i$ has received responses. The initial value of *requested-procs* is $\emptyset$.

The variables local to each process P_i are as follows:

—*ready-set* : set of pairs (I, R) where R is the set of roles of interaction I that P_i is ready to assume in its current enrolement phase. It is reset to $\emptyset$ when P_i leaves its enrolement phase. When P_i activates $Coord_i$, it provides $Coord_i$ with *ready-set* as well as its entry time.

—et_i : the current entry time of P_i.

—*captor-info* : a pair $(Coord_j, t)$ meaning that P_i is captured by $Coord_j$ which has entry time t. The pair is reset to $(nil, 0)$ if P_i is not currently captured by any coordinator.

—*request-queue* : queue of capture requests currently under consideration by P_i, i.e., granted or deferred. It is implemented as a priority queue of triples $(Coord_j, t, I)$ with the highest priority given to the triple with the smallest t. A triple $(Coord_j, t, I) \in$ *request-queue* means that $Coord_j$, with entry time t, has sent a capture request to P_i; I is the interaction for which $Coord_j$ is building a quorum. Triple $(Coord_j, t, I)$ is deleted from the queue when $Coord_j$'s capture request is rejected by P_i or is withdrawn by $Coord_j$. The initial value of *request-queue* is $\emptyset$.

—*yet-to-reply-coord* : set of coordinators to whose capture requests P_i has not yet responded. The initial value of *yet-to-reply-coord* is $\emptyset$.

Furthermore, `exist-quorum`(Q, I) is a Boolean function that determines whether set Q contains a quorum of an interaction I. We discuss how to compute such a function in Section 5.

The algorithm is given in Figures 2 and 3 as two CSP-like repetitive commands, one describing the behavior of a coordinator, and the other the behavior of a process. Message passing between coordinators and processes is buffered. The **send** command deposits a message in an input buffer associated with the receiving process. The **receive** command removes a message of a desired type from the buffer. We assume that messages sent by the same process are received in the order sent.

Both of the repetitive commands are executed concurrently with the local code of each process (the code a process executes when it is in its local computing phase). The repetitive commands are of the form

$$*[gc_1 \;\Box\; gc_2 \;\Box\; \ldots \Box\; gc_k]$$

where each gc_i is a guarded command. The guarded commands, which we also referred to as *rule*s, are numbered for ease of reference.

The guard of a rule consists of a conjunction of Boolean expressions and a **receive** command; both parts are optional. A rule can be executed only if it is *enabled*: the Boolean part of its guard (if any) evaluates to true, and a message of the desired type (if any) can be found in the input buffer. If more than one rule is enabled at a time, then one of them will be chosen for execution, and the choice is nondeterministic. We do, however, assume weak fairness, i.e., a rule that is continually enabled is eventually executed.

Note that the rules for $Coord_i$ and P_i could be merged together into a single repetitive construct to be executed by P_i. In this case, there would be no need to introduce auxiliary coordinators into the system. Separating out the role of the coordinators, however, makes the algorithm easier to understand.

In Rule 2.0, by setting *captor-info* to $(Coord_i, et_i)$, P_i is captured by $Coord_i$ immediately after entering its enrolement phase and activating $Coord_i$. When $Coord_i$ is activated (Rule 1.0), it applies Rule 1.1 to choose an interaction for which to build a quorum. When an interaction has been chosen (i.e., $Int \neq nil$), $Coord_i$ initializes its partial quorum *p-quorum* to contain the pairs (r, P_i) for each role r of the interaction that P_i is ready to assume.

Note that in Rule 1.1, $Coord_i$ assumes that P_i has been captured prior to the execution of the rule (because, in general, placing $(r, P_j) \in$ *p-quorum* means that P_j has been captured by $Coord_i$). In fact, our algorithm guarantees that after the execution of Rules 1.0 and 2.0, P_i remains captured by $Coord_i$ (i.e., *captor-info* $= (Coord_i, et_i)$ in P_i) until $Coord_i$ has stopped its quorum-building activity (i.e., $Coord_i$'s *active* $=$ false). This is because when a coordinator $Coord_j$ attempts to capture P_i, if $Coord_j$ is older than P_i, then $Coord_j$ is not allowed to capture P_i. Moreover, if $Coord_j$ is younger, then $Coord_i$ (which has the same age as P_i) wins the contention and retains P_i.

After choosing an interaction in Rule 1.1, $Coord_i$ begins to capture processes to fill the roles (Rule 1.2) until it has built a quorum (Rule 1.3) or it has determined that it cannot build any quorum (Rule 1.4) for the chosen interaction. In Rule 1.4 the Boolean condition `failed` indicates whether or not $Coord_i$'s attempt to build a quorum of *Int* has failed, and is defined as a disjunction of two clauses as follows:

```
*[  1.0 receive Start(ready-set, et_i) from P_i  -->  /*  P_i activates Coord_i. */
      ∀ (I, R) ∈ ready-set,  chosen(I) := false ;
      active := true ;

□ 1.1 active ; Int = nil  -->
/*  Choose a new interaction for which to build a quorum.  If all interactions have been  */
/*  previously chosen, then Coord_i releases P_i and ceases all quorum-building activity.  P_i  */
/*  then waits to be captured by another coordinator.                                          */

      [ ∃ (I, R) ∈ ready-set, ¬chosen(I)  -->
            let (I, R) ∈ ready-set be such that ¬chosen(I) ;
            chosen(I) := true ;
            Int := I ;
            p-quorum := { (r, P_i) | r ∈ R } ;
            requested-procs := { P_i } ;
      □ ∀ (I, R) ∈ ready-set, chosen(I)  -->
            send Abort to P_i ;
            requested-procs := ∅ ;
            active := false ;  ]

□ 1.2 active; Int ≠ nil ; target = nil ; ¬ exist-quorum (p-quorum , Int) ; ¬ failed -->
/*  Select a new process to which to send a capture request. */
      [ 𝒫(Int) = requested-procs  -->  skip
      □ ∃ r ∈ ℛ(Int), (∀ P_j ∈ 𝒫(r),  (r, P_j) ∉ p-quorum ) ⋀ 𝒫(r) ⊄ requested-procs  -->
            let r ∈ ℛ(Int) be such a role in the above condition ;
            let target be a process in 𝒫(r) − requested-procs ;
      □ else  -->  let target be a process in 𝒫(Int) − requested-procs ; ]
      [ target  ≠ nil  -->  send Request(et_i, Int) to target ;
      □ else  -->  skip ;  ]

□ 1.3 active; Int ≠ nil ; target = nil ; exist-quorum (p-quorum , Int)  -->
/*  Coord_i has successfully built a quorum (which must involve P_i). It releases the processes  */
/*  involved in the quorum and informs them of their respective roles. Also, Coord_i releases  */
/*  any additional processes it may have captured and withdraws its deferred capture requests.  */
/*  Coord_i then ceases all quorum-building activity.                                          */

      let Q ⊆ p-quorum  be a quorum of Int ;
      for (r, P_j) ∈ Q do send Success(Int, r) to P_j ;
      for P_j ∈ 𝒫(Int) such that (∃ r, (r, P_j) ∈ (p-quorum ⋃ d-quorum)) ⋀ (∀ r, (r, P_j) ∉ Q) do
            send Abort to P_j ;
      p-quorum := ∅ ;
      d-quorum := ∅ ;
      Int:= nil ;
      active:= false ;

□ 1.4 active; Int ≠ nil ; target = nil ; failed  -->
/*  Coord_i's attempt to build a quorum of Int has failed.  It releases all captured processes  */
/*  except P_i and withdraws all deferred capture requests.  After this rule is executed, Rule 1.1  */
/*  will be enabled because Int = nil and active remains true.                                  */

      for P_j, P_j ≠ P_i, such that ∃r, (r, P_j) ∈ (p-quorum ⋃ d-quorum) do send Abort to P_j ;
      p-quorum := ∅ ;
      d-quorum := ∅ ;
      Int := nil ;
```

Fig. 2. The EG algorithm for coordinator $Coord_i$.

□ 1.5 **receive** $Yes(t, I, R)$ **from** P_j $\longrightarrow$
/* P_j grants $Coord_i$'s capture request and informs $Coord_i$ of the set R of roles it is ready to assume. */

[$\exists (r, P_j) \in$ *d-quorum* ; $t = et_i$; $I = Int$ $\longrightarrow$
for $r \in R$ **do**
[*p-quorum* := *p-quorum* $\bigcup \{ (r, P_j) \}$;
d-quorum := *d-quorum* $- \{ (r, P_j) \}$;]
□ P_j = *target* ; $t = et_i$; $I = Int$ $\longrightarrow$
for $r \in R$ **do** *p-quorum* := *p-quorum* $\bigcup \{ (r, P_j) \}$;
requested-procs := *requested-procs* $\bigcup$ {*target*};
target := *nil* ;
□ **else** $\longrightarrow$ **skip** ; /* out-of-date message */]

□ 1.6 **receive** $No(t, I)$ **from** P_j $\longrightarrow$
/* P_j rejects $Coord_i$'s capture request. */
[$\exists (r, P_j) \in$ *d-quorum* ; $t = et_i$; $I = Int$ $\longrightarrow$
for $(r, P_j) \in$ *d-quorum* **do** *d-quorum* := *d-quorum* $- \{ (r, P_j) \}$;
□ P_j = *target* ; $t = et_i$; $I = Int$ $\longrightarrow$
requested-procs := *requested-procs* $\bigcup$ {*target*};
target := *nil* ;
□ **else** $\longrightarrow$ **skip** ; /* out-of-date message */]

□ 1.7 **receive** $Deferred(I, R)$ **from** P_j $\longrightarrow$
/* P_j informs $Coord_i$ of the roles P_j is willing to assume. $Coord_i$'s capture request has been deferred by P_j, however. */

for $r \in R$ **do** *d-quorum* := *d-quorum* $\bigcup \{ (r, P_j) \}$;
requested-procs := *requested-procs* $\bigcup$ {*target*};
target:= *nil* ;

□ 1.8 **receive** *Switch* **from** P_j $\longrightarrow$
/* P_j requests to be released by $Coord_i$ because of a conflict with an older coordinator. */

[¬ `exist-quorum` (*p-quorum* , *Int*); $\exists (r, P_j) \in$ *p-quorum* $\longrightarrow$
send *Switch_ok* **to** P_j ;
for $(r, P_j) \in$ *p-quorum* **do**
[*p-quorum* := *p-quorum* $- \{ (r, P_j) \}$;
d-quorum := *d-quorum* $\bigcup \{ (r, P_j) \}$;]
□ **else** $\longrightarrow$ **skip** ; /* out-of-date message */]
]

Fig. 2. The EG algorithm for coordinator $Coord_i$ (continued).

$*[$ 2.0 transit from a local computing phase to an enrolement phase $\longrightarrow$
/* P_i activates $Coord_i$ and provides it with *ready-set* and et_i. */

$\quad$ *ready-set* $:= \{ (I,R) \mid P_i$ is ready to assume any one of the set R of roles of $I \}$;
$\quad$ obtain a new entry time et_i from P_i's local clock ;
$\quad$ **send** $Start(\textit{ready-set}, et_i)$ **to** $Coord_i$;
$\quad$ *captor-info* $:= (Coord_i, et_i)$; $\qquad$ /* P_i is now captured by $Coord_i$. */

$\Box$ 2.1 **receive** $Request(et_j, I)$ **from** $Coord_j$ $\longrightarrow$
/* If P_i is not ready to enrole into any role of I or P_i is younger than $Coord_j$, then P_i rejects */
/* $Coord_j$'s capture request. Otherwise, if P_i is not captured by any coordinator, P_i will */
/* reply to $Coord_j$'s capture request when rule 2.5 is next executed. If P_i is captured by */
/* $Coord_k$, then if $Coord_j$ is younger than $Coord_k$, P_i defers $Coord_j$'s request, and if $Coord_j$ */
/* is older, P_i sends message *Switch* to $Coord_k$ to resolve the conflict. */

$\quad$ [$et_i < et_j$; $\exists\, R,\ (I,R) \in \textit{ready-set}$ $\longrightarrow$
$\qquad$ *request-queue* $:=$ *request-queue* $\bigcup \{ (Coord_j, et_j, I) \}$;
$\qquad$ **let** *captor-info* $= (Coord_k, et_k)$;
$\qquad$ [$Coord_k = nil$ $\longrightarrow$
$\qquad\quad$ *yet-to-reply-coord* $:=$ *yet-to-reply-coord* $\bigcup \{ Coord_j \}$;
$\qquad$ $\Box$ $Coord_k \neq nil$; $et_j > et_k$ $\longrightarrow$
$\qquad\quad$ **let** R be such that $(I,R) \in$ *ready-set* ;
$\qquad\quad$ **send** $Deferred(I,R)$ **to** $Coord_j$;
$\qquad$ $\Box$ $Coord_k \neq nil$; $et_j < et_k$ $\longrightarrow$
$\qquad\quad$ *yet-to-reply-coord* $:=$ *yet-to-reply-coord* $\bigcup \{ Coord_j \}$;
$\qquad\quad$ **send** *Switch* **to** $Coord_k$;]
$\quad$ $\Box$ **else** $\longrightarrow$ **send** $No(et_j, I)$ **to** $Coord_j$;]

$\Box$ 2.2 **receive** $Success(I, r)$ **from** $Coord_j$ $\longrightarrow$
/* The captor $Coord_j$ informs P_i that a quorum of I in which P_i assumes role r has been */
/* established. P_i now rejects all deferred capture requests. */

$\quad$ **for** $(Coord_k, et_k, I') \in$ *request-queue*, $Coord_k \neq Coord_j$, **do send** $No(et_k, I')$ **to** $Coord_k$;
$\quad$ *yet-to-reply-coord* $:= \emptyset$;
$\quad$ *captor-info* $:= (nil, 0)$;
$\quad$ *request-queue* $:= \emptyset$;
$\quad$ *ready-set* $:= \emptyset$;
$\quad$ enrole into role r of I ;
$\quad$ leave the current enrolement phase ;

$\Box$ 2.3 **receive** *Abort* **from** $Coord_j$ $\longrightarrow$
/* $Coord_j$ withdraws its capture request and releases P_i if P_i is currently captured by $Coord_j$. */

$\quad$ [$\exists\, et_j, I,\ (Coord_j, et_j, I) \in$ *request-queue* $\longrightarrow$
$\qquad$ **let** $(Coord_j, et_j, I)$ be such that $(Coord_j, et_j, I) \in$ *request-queue* ;
$\qquad$ *request-queue* $:=$ *request-queue* $- \{ (Coord_j, et_j, I) \}$;
$\qquad$ [$Coord_j \in$ *yet-to-reply-coord* $\longrightarrow$
$\qquad\quad$ *yet-to-reply-coord* $:=$ *yet-to-reply-coord* $- \{ Coord_j \}$;
$\qquad$ $\Box$ **else** $\longrightarrow$ **skip** ;]
$\qquad$ **let** *captor-info* be $(Coord_k, et_k)$;
$\qquad$ [$Coord_k = Coord_j$ $\longrightarrow$ *captor-info* $:= (nil, 0)$;
$\qquad$ $\Box$ **else** $\longrightarrow$ **skip** ;]
$\quad$ $\Box$ **else** $\longrightarrow$ **skip** ; $\quad$ /* out-of-date message */]

Fig. 3. The EG algorithm for process P_i.

□ 2.4 **receive** *Switch_ok* **from** $Coord_j$ $\longrightarrow$

/* The captor $Coord_j$ releases P_i to resolve a conflict; $Coord_j$'s capture request is now de- */
/* ferred by P_i. */

$captor\text{-}info := (nil, 0)$;

□ 2.5 $captor\text{-}info = (nil, 0)$; $request\text{-}queue \neq \emptyset$ $\longrightarrow$

/* Choose the oldest coordinator from those that have requested P_i for capture as the new */
/* captor. All capture requests sent by the other processes in *yet-to-reply-coord* are now */
/* answered with *Deferred* messages. */

let $(Coord_j, et_j, I) \in request\text{-}queue$ be such that
$\forall\ (Coord_k, et_k, I') \in request\text{-}queue,\ et_j \leq et_k$;
$captor\text{-}info := (Coord_j, et_j)$;
let R be such that $(I, R) \in$ *ready-set* ;
send $Yes(et_j, I, R)$ **to** $Coord_j$;
$yet\text{-}to\text{-}reply\text{-}coord := yet\text{-}to\text{-}reply\text{-}coord - \{\ Coord_j\ \}$;
for $Coord_k \in yet\text{-}to\text{-}reply\text{-}coord$ **do**
[**let** et_k, I' be such that $(Coord_k, et_k, I') \in request\text{-}queue$;
let R' be such that $(I', R') \in$ *ready-set* ;
send $Deferred(I', R')$ **to** $Coord_k$;
$yet\text{-}to\text{-}reply\text{-}coord := yet\text{-}to\text{-}reply\text{-}coord\ - \{\ Coord_k\ \}$;]

]

Fig. 3. The EG algorithm for process P_i (continued).

$$\begin{aligned}\texttt{failed} \equiv (\exists\, r \in \mathcal{R}(Int)\ \forall\, P_j \in \mathcal{P}(r),\ P_j \in requested\text{-}procs \\ \bigwedge\ (r, P_j) \notin (p\text{-}quorum \bigcup d\text{-}quorum)) \quad \bigvee \\ (\mathcal{P}(Int) = requested\text{-}procs \bigwedge \neg\ \texttt{exist-quorum}((p\text{-}quorum \bigcup d\text{-}quorum), Int))\end{aligned}$$

The first disjunct of `failed` decides whether there exists a role r for which $Coord_i$ cannot find a process. That is, every process $P_j \in \mathcal{P}(r)$ has either rejected capture by $Coord_i$ or is not willing to assume r (but is ready to assume some other role of the interaction *Int*). The other disjunct concerns the case in which all potential enrolers of the interaction have been requested by $Coord_i$ for capture, but $Coord_i$ cannot reach a quorum even if all its deferred capture requests were granted.

Note that the first clause allows "early stopping" of $Coord_i$'s quorum-building activity as soon as it becomes clear that some role of *Int* cannot be filled, even if $\mathcal{P}(Int) \neq requested\text{-}procs$. Moreover, if we do not allow a process to be ready for more than one role of the same interaction in any enrolement phase, then the second clause implies the first. In this case, therefore, only the first clause is needed to detect failure. Otherwise, it may be the case that the second clause is true and the first is false, and, therefore, both clauses are needed. For example, consider an interaction I having three roles r_1, r_2 and r_3, where $\mathcal{P}(r_1) = \{P_1\}$, $\mathcal{P}(r_2) = \{P_2, P_3\}$ and $\mathcal{P}(r_3) = \{P_2, P_3\}$. Suppose $Coord_1$ is building a quorum of I and has captured P_1 for role r_1 and P_2 for both r_2 and r_3 (i.e., P_2 is willing to assume either role r_2 or r_3). If $Coord_1$'s capture request has been rejected by P_3, then $Coord_1$ has requested all three processes for capture, but its partial quorum $\{(r_1, P_1), (r_2, P_2), (r_3, P_2)\}$ does not contain a quorum. In the partial quorum, however, for each role of I there exists a captured process ready to assume the role.

Conditions $\neg$`exist-quorum`(*p-quorum*, *Int*) and $\neg$`failed` in the guard to Rule 1.2 mean that $Coord_i$'s attempt to build a quorum has not yet succeeded nor failed. In this case $Coord_i$ can proceed *conservatively*—by waiting for all of its deferred capture requests to be granted or rejected, or *aggressively*—by sending a new capture request to a process that has not been previously requested, hoping that the process is ready to fill a role that would assist $Coord_i$ in reaching a quorum. The choice constitutes a fine tuning of the algorithm: the conservative approach may result in fewer messages, while the aggressive approach may give better response time. In Rule 1.2 we have opted for the aggressive approach; for the conservative approach, the condition "$d\text{-}quorum = \emptyset$" would also appear in the guard of Rule 1.2.

In Rule 1.2 we consider three cases in choosing a new process to capture. First, if $\mathcal{P}(Int) = requested\text{-}procs$ then all potential enrolers of *Int* have been requested for capture. Given that `failed` = false, $\mathcal{P}(Int) = requested\text{-}procs$ implies that $(p\text{-}quorum \bigcup d\text{-}quorum)$ contains a quorum of *Int*. Thus, $Coord_i$ can only wait hoping that all its deferred capture requests will eventually be granted so that a quorum can be established. For the second and third cases, we have that $\mathcal{P}(Int) \neq requested\text{-}procs$. In the second case, there exists a role r which no process captured by $Coord_i$ is willing to assume, but there is a potential enroler of r to which $Coord_i$ has not yet sent a capture request. So $Coord_i$ can choose this potential enroler to send a capture request. In the third case, for every role of the interaction *Int* there is a process captured by $Coord_i$ which is ready to assume the role but $Coord_i$ has not yet been able to reach a quorum. (There must be some captured process which is ready to assume more than one role.) Since $\mathcal{P}(Int) \neq requested\text{-}procs$, $Coord_i$ can try to capture a new process to enlarge its partial quorum, hoping that each role will be filled by a distinct process. In the algorithm we let $Coord_i$ choose an arbitrary process from those that are not yet requested. Of course, $Coord_i$ may behave more intelligently by first determining a role such that finding a new process for the role would help $Coord_i$ to establish a quorum. However, as we shall see in Section 5, finding such a role is closely related to the problem of Maximum Matching and is time-consuming.

Rules 1.5, 1.6, and 1.7 process P_j's responses to $Coord_i$'s capture requests, and Rule 1.8 resolves conflicts between coordinators. In the EG algorithm, $Coord_i$ sends only one capture request to each process it wishes to capture. The capture request may be granted (Rule 1.5), rejected (Rule 1.6), or deferred (Rule 1.7). Note that if $Coord_i$ has sent a capture request to P_j, $Coord_i$ may receive from P_j at most one *No* message and at most one *Deferred* message. However, $Coord_i$ may receive from P_j more than one *Yes* message and more than one *Switch* message. For example, suppose P_j first responds to $Coord_i$'s capture request with a *Deferred* message. If P_j is released by its captor, it may send *Yes* to $Coord_i$ informing $Coord_i$ that its deferred capture request is now granted. If before $Coord_i$ has established a quorum an older coordinator $Coord_k$ attempts to capture P_j, then P_j sends *Switch* to $Coord_i$ asking for its release. When $Coord_i$ has released P_j (Rule 1.8), P_j may again later send *Yes* to $Coord_i$ if P_j is released by $Coord_k$.

Rules 2.1 to 2.6 describe the behavior of P_i in response to capture requests from coordinators. On receiving a capture request $Request(et_j, I)$ from $Coord_j$ (Rule 2.1), if P_i is not ready to enrole in I or P_i is younger than $Coord_j$, then P_i rejects the request by $Coord_j$. Otherwise, there are two cases. If P_i is not captured by any coordinator, then P_i adds $Coord_j$ to *yet-to-reply-coord*, and will respond to $Coord_j$'s request when Rule 2.5 is next executed. If P_i is captured by some coordinator $Coord_k$, then P_i compares the ages of the two coordinators $Coord_j$ and $Coord_k$. If $Coord_k$ is older, then P_i remains captured by $Coord_k$ and $Coord_j$'s request is deferred. If $Coord_j$ is older, then P_i sends *Switch* to $Coord_k$ asking for its release. P_i also adds $Coord_j$ to *yet-to-reply-coord* since it cannot answer $Coord_j$'s request until the conflict between $Coord_j$ and $Coord_k$ has been resolved. When P_i receives the confirmation *Switch_ok* from $Coord_k$ (Rule 2.4), it then executes Rule 2.5 to select a new captor. At that time if $Coord_j$ is still the oldest coordinator among those that have requested P_i for capture, $Coord_j$ will be the new captor of P_i and its capture request to P_i will be granted. If, however, some other coordinator older than $Coord_j$ has also requested P_i for capture while the conflict between $Coord_j$ and $Coord_k$ is ongoing, then $Coord_j$ will not be the new captor. Instead, P_i will respond to $Coord_j$'s capture request with a *Deferred* message.

Due to the asynchrony of the system, messages of types *Yes*, *No*, *Switch*, and *Abort* may become "out-of-date" and therefore must be discarded by the receiver to maintain correctness of the EG algorithm [Joung and Smolka 1990]. For example, suppose coordinator $Coord_i$ has determined that it cannot successfully establish a quorum and consequently has issued an *Abort* message to P_j to withdraw its deferred capture request. At the same time P_j decides to reject $Coord_i$'s deferred capture request (because it has been placed in an established quorum). Then, the *Abort* message to P_j (Rule 2.3) and the *No* message to $Coord_i$ (Rule 1.6) will be out-of-date when they arrive at P_j and $Coord_i$, respectively. The situation is similar if P_j instead grants $Coord_i$'s deferred capture request by sending it a *Yes* message (Rule 1.5).

The entry times and interaction names carried by *Yes* and *No* messages allow the receiver to tell if they are out-of-date, while no additional information is required in *Switch* or *Abort* messages for this purpose. An *Abort* message from $Coord_i$ to P_j is out-of-date if there is no deferred capture request by $Coord_i$ in P_j's request queue (Rule 2.3). A *Switch* message from P_j to its captor $Coord_i$ is out-of-date if $Coord_i$ has already established a quorum or already released P_j (Rule 1.8).

3.3 An Example

To illustrate the behavior of the EG algorithm, consider an interaction I with three roles r_1, r_2, and r_3. Assume that P_1 and P_2 are ready for role r_1, P_3 and P_4 for r_2, and P_5 and P_6 for r_3, as shown below:

$$I :: \begin{cases} r_1 & : P_1, P_2 \\ r_2 & : P_3, P_4 \\ r_3 & : P_5, P_6 \end{cases}$$

Furthermore, the ages of the six processes are in a decreasing order such that P_6 is the youngest. In the following we give a possible scenario of events for the six processes executing the EG algorithm:

—Initially, each $Coord_i$ has captured P_i, $1 \leq i \leq 6$.

—$Coord_1$ requests capture of P_3 but is rejected. It then issues a request to P_4 but again is rejected because P_1 (and thus $Coord_1$) is older than both P_3 and P_4.

$Coord_1$ then releases P_1 and idles. Similarly, $Coord_2$ releases P_2 and idles. Also, $Coord_3$ releases P_3 and idles after being rejected by both P_5 and P_6.

—$Coord_4$ captures P_1.

—$Coord_5$ and $Coord_6$ both request capture of P_4 which is captured by $Coord_4$. Their requests are deferred by P_4 as $Coord_5$ and $Coord_6$ are younger than $Coord_4$.

—$Coord_4$'s capture requests to capture P_5 and P_6 are rejected. It releases P_1 and P_4 and then idles.

—P_4 grants $Coord_5$'s capture request.

—$Coord_6$ captures P_1.

—$Coord_5$ requests capture of P_1, which belongs to $Coord_6$. Since $Coord_5$ is older than $Coord_6$, $Coord_5$ captures P_1 and $Coord_6$'s capture request is now deferred by P_1. $Coord_5$ then has established a quorum involving P_1, P_4 and P_5 (among which P_5 is the youngest process).

—After capturing P_2 and P_3, $Coord_6$ then has established another quorum of I (among which P_6 is the youngest process).

4. ANALYSIS OF THE EG ALGORITHM

In this section we prove that our algorithm satisfies the Enrolement Guard Scheduling Problem safety and liveness properties defined in Section 2, and analyze its message complexity.

THEOREM 1 (SAFETY). *At any time the established quorums are pairwise disjoint.*

PROOF. This is a consequence of the fact that a coordinator can establish a quorum only if it has captured all of the processes involved in the quorum, and that a process can be captured by only one coordinator at a time. □

The following lemmas are used to establish the liveness result.

LEMMA 4.1. *If coordinator $Coord_i$ chooses to build a quorum for interaction I and issues a capture request to P_j, then eventually one of the following must occur: (1) P_j rejects $Coord_i$'s capture request; (2) P_j defers the request and $Coord_i$ later withdraws it; or (3) P_j is captured by $Coord_i$ and remains captured until $Coord_i$ stops (successfully or unsuccessfully) building a quorum for I and releases P_j.*

PROOF. P_j rejects $Coord_i$'s capture request because of either one of the following: (a) P_j is younger than $Coord_i$ or P_j is not ready to enrole into I (Rule 2.1), or (b) P_j has been placed in an established quorum (Rule 2.2). $Coord_i$ withdraws its capture request to P_j when $Coord_i$ aborts its quorum-building activity for I. If $Coord_i$'s capture request is not rejected by P_j or withdrawn by $Coord_i$, then it is either granted or deferred by P_j. Note that a capture request which has been granted may later need to be deferred in order to resolve a conflict.

We claim that if P_j does not reject $Coord_i$'s capture request and $Coord_i$ does not withdraw its capture request, then eventually P_j will be captured by $Coord_i$ and remain captured until $Coord_i$ has finished its quorum-building activity for I and released P_j.

Suppose to the contrary that our claim does not hold. Then, there are two possibilities: (1) P_j defers $Coord_i$'s capture request infinitely often; or (2) P_j forever maintains $Coord_i$'s capture request as deferred.

If P_j defers $Coord_i$'s capture request infinitely often, then there must exist a coordinator $Coord_k$ that attempts to build a quorum an infinite number of times. Furthermore, in each attempt, $Coord_k$ (or, more precisely, P_k) obtains an entry time that is smaller than the entry time $Coord_i$ currently possesses (i.e., $Coord_k$ is older than $Coord_i$), and issues a capture request to P_j while P_j is captured by $Coord_i$. Such a request causes P_j to re-defer $Coord_i$'s capture request. Assuming that P_k's logical clock is initially set to zero, we have a contradiction, since the sequence of entry times obtained by a process is strictly increasing, with consecutive entry times being at least one apart.

If P_j forever maintains $Coord_i$'s capture request as deferred, then there must exist a coordinator $Coord_k$ older than $Coord_i$ such that $Coord_k$ is building a quorum and has captured P_j forever. $Coord_k$ cannot finish its quorum-building activity and release P_j because similarly one of $Coord_k$'s capture requests has been maintained forever as deferred by a process P_l. Continuing in this fashion, we have an infinite sequence of coordinators with strictly increasing ages, a contradiction to the fact that there is only a finite number of coordinators in the system. □

LEMMA 4.2. *Upon being activated, a coordinator eventually either succeeds in establishing a quorum or completes its quorum-building activity without establishing any quorum.*

PROOF. Upon being activated, a coordinator $Coord_i$ chooses an interaction I, in which P_i is ready to enrole, to attempt to build a quorum (i.e., $Int = I$ in Rule 1.1). We shall argue that either Rule 1.3 is eventually executed, i.e., $Coord_i$ succeeds in establishing a quorum for I, or Rule 1.4 is eventually executed, i.e., $Coord_i$'s attempt to establish a quorum for I fails. In the latter case, $Coord_i$ chooses another untried interaction that P_i is ready for, should one exist. By applying this argument to each of $Coord_i$'s attempts, the desired lemma follows.

We begin by noting that the conditions *active* and $Int \neq nil$ in the Boolean guards of Rules 1.3 and 1.4 remain true throughout $Coord_i$'s attempt to build a quorum for I. The other condition these two guards have in common is $target = nil$. The negation of this condition, $target \neq nil$, indicates that $Coord_i$ has issued a new capture request to some process, and is waiting for a response (i.e. a *Yes*, *No*, or *Deferred* message) from that process. By Lemma 4.1, such a response is eventually received by $Coord_i$, after which $target = nil$ is satisfied.

We are left to consider condition `exist-quorum`$(p\text{-}quorum, I)$ of Rule 1.3 and condition `failed` of Rule 1.4. We show that exactly one of them eventually holds. Since `failed` and `exist-quorum`$(p\text{-}quorum, I)$ cannot hold simultaneously, it suffices to show that eventually $\neg$`failed` implies `exist-quorum`$(p\text{-}quorum, I)$. By definition, $\neg$`failed` implies

$$\mathcal{P}(I) \neq requested\text{-}procs \ \bigvee\ \texttt{exist-quorum}((p\text{-}quorum \bigcup d\text{-}quorum), I)$$

where the first disjunct means that there is a process in $\mathcal{P}(I)$ $Coord_i$ has not yet requested for capture, and the other disjunct means `exist-quorum`$(p\text{-}quorum, I)$ or $d\text{-}quorum \neq \emptyset$ (i.e., $Coord_i$ has successfully built a quorum or at least one of its capture requests has been deferred).

Given that $\neg$`exist-quorum`$(p\text{-}quorum, I)$, $\neg$`failed`, and $\mathcal{P}(I) \neq requested\text{-}procs$, Rule 1.2 will eventually be executed thereby letting $Coord_i$ issue a new capture request to a previously untried process. Since the number of processes in $\mathcal{P}(I)$ is finite, eventually all processes will have been requested for capture, at which point $\neg$`failed` implies `exist-quorum`((*p-quorum* $\bigcup$ *d-quorum*), I). By Lemma 4.1, eventually *d-quorum* $= \emptyset$, at which point $\neg$`failed` implies `exist-quorum`(*p-quorum*, I). Therefore, Rule 1.3 or Rule 1.4 is eventually enabled. It can be seen that once either one of these rules is enabled, it remains enabled until executed. By the weak fairness assumption on the execution of rules, Rule 1.3 or Rule 1.4 is eventually executed. □

LEMMA 4.3. *Suppose that $Coord_i$, when possessing entry time t_i, issues a capture request to P_j. Then, when P_j enters its next enrolement phase after receiving the capture request, its new entry time will be greater than t_i.*

PROOF. The capture request from $Coord_i$ to P_j has a timestamp larger than or equal to t_i. When P_j receives this capture request, it advances its logical clock to be larger than the timestamp of the request. So when P_j enters its next enrolement phase, the new entry time it obtains from its logical clock will be greater than t_i. □

THEOREM 2 (LIVENESS). *If there exists a quorum then eventually some process involved in the quorum will participate in an established quorum.*

PROOF. Suppose that at some point there exists a set $\mathcal{Q}$ of quorums (which may not be pairwise disjoint). We define the *age* of a quorum to be the age of the youngest process involved in the quorum. We shall argue that the oldest quorum in $\mathcal{Q}$ will eventually cease to exist due to the fact that some process involved in the quorum is placed in an established quorum. By repeatedly applying the argument to the remaining quorums of $\mathcal{Q}$, we prove that all the quorums in $\mathcal{Q}$ will eventually cease to exist, thus proving the theorem.

Consider the oldest quorum Q in $\mathcal{Q}$ (if there is more than one such quorum, then consider an arbitrary one of them), and let Q be a quorum of interaction I. By the definition of a quorum, each P_i involved in Q is in an enrolement phase, and we will denote this enrolement phase by e_i.[5] Furthermore, let P_m be the youngest process involved in Q (thus the age of Q is the age of P_m).

When P_m enters enrolement phase e_m, its coordinator $Coord_m$ attempts to build a quorum for each interaction in which P_m is ready to enrole until $Coord_m$ has successfully established a quorum (Rule 1.3), or has tried out all the interactions (Rule 1.1). By Lemma 4.2, $Coord_m$ eventually either succeeds in establishing a quorum or completes its quorum-building activity without establishing any quorum for any interaction.

If $Coord_m$ successfully establishes a quorum, say Q', then Q' must contain P_m, since $Coord_m$ can only capture P_m and processes that are older than P_m (i.e., the age of the established quorum must be equal to or older than the age of P_m), and the age of the oldest quorum(s) in $\mathcal{Q}$ *is* the age of P_m. Since P_m has been placed in an established quorum, Q ceases to exist. Note that if the oldest quorum in $\mathcal{Q}$ is unique, then $Q' = Q$.

If $Coord_m$ completes without establishing any quorum, then it must have attempted to build a quorum of I and failed. By the definition of the Boolean guard

[5]The various enrolement phases through which a process P_i passes can be distinguished by their entry times. Thus, e_i can be thought of as the entry time P_i obtains upon entering the enrolement phase in which Q exists.

`failed` and Rule 1.4, some process involved in Q, say P_i, must have rejected $Coord_m$'s capture request. Note that $P_i \neq P_m$ because when P_m enters its enrolement phase, it activates $Coord_m$ and is then immediately captured by $Coord_m$ (Rule 2.0). Until $Coord_m$ completes its quorum-building activity and releases P_m, no other coordinator can capture P_m: an older coordinator will fail to capture P_m due to the policy that a coordinator is only allowed to capture older processes; a younger coordinator will lose out to $Coord_m$ due to the policy that conflicts are resolved in favor of older coordinators.

P_i rejects $Coord_m$'s capture request either because (1) it is younger than $Coord_m$ or it is not ready to enrole into I (Rule 2.1), or (2) it has been placed in an established quorum (Rule 2.2). Since P_i in enrolement phase e_i is older than $Coord_m$, (1) implies that P_i has not yet entered e_i or has already left e_i (after having participated in some established quorum) when it receives $Coord_m$'s capture request. The former case can be ruled out since, by Lemma 4.3, P_i would be younger than $Coord_m$ upon entering e_i. We thus have that P_i has already entered and left e_i and, in both (1) and (2), quorum Q ceases to exist. □

In the above proof we have shown that when $Coord_m$ completes its quorum-building activity, it either successfully establishes a quorum involving P_m, or completes without establishing any quorum due to the fact that some process P_i involved in Q has been placed in an established quorum; in either case Q ceases to exist. The observation that $P_i \neq P_m$ implies that no other coordinator could have successfully captured all processes of Q and established Q while $Coord_m$ was building a quorum. Hence, if Q is established, it must be established by $Coord_m$. We thus have the following corollary.

COROLLARY 4.4. *Any quorum established by a coordinator $Coord_m$ must involve P_m. In fact, P_m is the youngest process in the quorum.*

To analyze the algorithm's message complexity, we first compute the number of messages that can be transmitted by $Coord_i$ itself or generated by other processes in response to $Coord_i$'s messages while $Coord_i$ is attempting to capture a process P_j. We consider a worst-case scenario.

(1) $Coord_i$ sends a *Request* message to P_j.

(2) P_j is captured by $Coord_k$ which is younger than $Coord_i$. So, P_j sends a *Switch* message to $Coord_k$.

(3) $Coord_k$ sends the confirmation *Switch_ok* to P_j. $Coord_k$'s capture request is now deferred by P_j.

(4) P_j sends a *Yes* message to $Coord_i$.

(5) $Coord_i$ establishes a quorum involving P_j. It sends a *Success* message to P_j.

(6) After P_j is placed into a quorum, P_j sends a *No* message to $Coord_k$, rejecting $Coord_k$'s deferred capture request.

A potential variation on the above scenario is the following:

(5′) $Coord_i$ fails to place P_j in a quorum and thus sends an *Abort* message to P_j.

(6′) P_j sends $Coord_k$ a *Yes* message to grant its deferred capture request.

In either worst-case scenario, $Coord_i$ generates (directly or indirectly) at most 6 messages in its attempt to recruit P_j to fill a role in an interaction.

LEMMA 4.5. *Let $Coord_i$ be a potential coordinator of interaction I, i.e., $P_i \in \mathcal{P}(I)$. Then, the maximum number of messages transmitted by $Coord_i$ itself or by other processes in response to $Coord_i$'s messages while $Coord_i$ is attempting to build a quorum of I is $2 + 6\,(|\mathcal{P}(I)| - 1)$.*

PROOF. While $Coord_i$ is attempting to build a quorum of I, it can capture at most $|\mathcal{P}(I)| - 1$ processes besides P_i. The capture of P_i takes only two messages (one by P_i to activate $Coord_i$, which then immediately proceeds to capture P_i, and another by $Coord_i$ to release P_i), and each capture of the $|\mathcal{P}(I)| - 1$ processes generates at most 6 messages. □

We now analyze the overall performance of the EG algorithm in terms of the number of messages needed to establish a quorum for an interaction.

THEOREM 3 (MESSAGE COMPLEXITY). *Let I be an interaction with $|\mathcal{R}(I)| = l$ and $|\mathcal{P}(I)| = m$, where $l \leq m$. Assume that there exists a quorum Q of I involving processes $P_1, \ldots, P_l$. Then, before Q ceases to exist, the total number of messages generated by $Coord_1$, …, $Coord_l$ in attempting to establish Q is no more than $4\,m^2 - 2m$.*

PROOF. When $Coord_i$, $1 \leq i \leq l$, attempts to establish a quorum of I, it may try to capture any of the m processes in $|\mathcal{P}(I)|$. By Lemma 4.5, $Coord_i$ can generate at most $2 + 6(m-1)$ messages, and, therefore, an upper bound on the number of messages to establish Q is:

$$l \times (2 + 6(m-1))$$

Due to the policy of only allowing a coordinator to capture older processes, the actual number of messages needed to establish Q is less than the above value, since an attempt to capture an older process will take at most 6 messages, and an attempt to capture a younger process will take 2 (one by the coordinator to request capture and the other by the process to reject the request).

Let P_j be the k_jth oldest process among the processes that are ready to enrole into I. When $Coord_j$ attempts to establish a quorum of I, it can generate at most $2 + 6(k_j - 1) + 2(m - k_j)$ messages: 2 messages to capture P_j, $6(k_j - 1)$ messages to capture processes older than P_j, and $2(m - k_j)$ messages in failed attempts at capturing younger processes. The maximum number of messages generated by coordinators $Coord_1$, …, $Coord_l$ to establish Q is thus:

$$\sum_{1 \leq i \leq l} 2 + 6(k_i - 1) + 2(m - k_i) \leq 4m^2 - 2m$$

□

Note that if we were to abandon our policy of only allowing a coordinator to capture older processes, then each of the m potential coordinators could generate $6\,m - 4$ messages (Lemma 4.5) and the message complexity of Theorem 3 would increase to:

$$m \times (6\,m - 4) = 6\,m^2 - 4\,m$$

Our EG algorithm is the first algorithm, to our knowledge, for coordinating first-order interactions. To allow comparison with existing coordination algorithms, we consider the message complexity of the EG algorithm in a zeroth-order setting. Specifically, consider a zeroth-order interaction I with m fixed participants. We first discuss the message complexity of Ramesh's efficient algorithm for coordinating I [Ramesh 1987]. By considering I as a first-order interaction with m roles, each of which can be assumed by only a fixed process, we are then able to draw an explicit comparison between Ramesh's algorithm and our own.

Ramesh's algorithm also allows each process P_i, when ready for an interaction I, to behave as the coordinator of I, which P_i does by attempting to capture all of the participants of I. Unlike the EG algorithm, however, P_i must capture the participants (which are fixed in the zeroth-order case) in the strict ordering of the process id's; i.e., it must capture P_j before P_k whenever $j < k$.

A coordinator can keep a captured process until some other process has rejected the coordinator's capture request. The order in which processes are captured guarantees that every capture request is eventually definitely answered, i.e., granted or rejected. Thus, a successful capture generates 3 messages, one message to request capture, one to grant the request, and another to release the captured process. An unsuccessful capture generates 2 messages, one to request capture and the other to reject the request.

Let I be a zeroth-order interaction with m participants. The worst-case scenario for Ramesh's algorithm, from a message standpoint, occurs when each of the first $m-1$ participants, acting as a coordinator, fails to capture the last participant (because it is not yet ready for the interaction) after having captured the first $m-1$ participants; and, finally, the last participant successfully captures all of the participants and establishes I. In this case, the total number of messages generated is

$$(m-1)\,(3(m-1)+2)+3m = 3m^2 - m + 1$$

We now analyze the complexity of the EG algorithm by treating I as a first-order interaction having m roles, each of which can only be assumed by a fixed process. Let P_i be the ith oldest participant in I. By the proof of Theorem 3, $Coord_i$ can generate at most $2+6(i-1)+2(m-i)$ messages, where $2(m-i)$ messages result from attempting to capture the $(m-i)$ processes younger than $Coord_i$.

Recall that the EG algorithm allows $Coord_i$ to abort its attempt to establish a quorum for I if it determines that there is a role for which it cannot capture any process to fill the role (see Rule 1.4 and the definition of the Boolean guard `failed` in Section 3.2). Since for each role of I there is only one process that can potentially assume the role, $Coord_i$ will stop its attempt to establish a quorum for I when any one of its capture requests is rejected. Therefore, the actual number of messages generated by $Coord_i$ is at most $2+6(i-1)+2$ for $i<m$, and is $2+6(m-1)$ for $i=m$. The maximum number of messages generated by the coordinators of I is thus

$$\Big(\sum_{i<m} 2+6(i-1)+2\,\Big) \;+\; (2+6(m-1)) = 3m^2+m-2$$

The EG algorithm compares favorably with Ramesh's zeroth-order algorithm in that it has the same asymptotic message complexity (quadratic) and, moreover, the same constant on the higher-order (quadratic) term.

5. MAXIMUM MATCHING AND SYMMETRIC ROLES

The activation of an instance of an interaction requires that each role is filled by a distinct process. A process in its enrolement phase, however, may be ready for different roles of the same interaction. Let *p-quorum* be the partial quorum of a coordinator $Coord_i$. To decide whether *p-quorum* contains a quorum, $Coord_i$ needs to find in *p-quorum* a subset Q of size $|\mathcal{R}(I)|$ such that no two elements in Q specify the same role or process. This is just an instance of the Maximum Matching problem[6] and can be solved in $O(m^{2.5})$ time [Hopcroft and Karp 1973], where m is the number of processes captured by $Coord_i$.

Clearly, we can avoid performing maximum matching and therefore reduce the time complexity of the coordinator's decision making process to linear if we do not allow a process to be ready to enrole in more than one role of the same interaction. In fact, in many multiparty applications certain roles of an interaction are *symmetric* in the sense that a process can enrole into any one of them without observing any difference. A process tries all of the symmetric roles because it does not know which of them are being targeted by other processes and wants to avoid being locked out.

For example, the recipient roles of the first-order bi-cast interaction (Section 1) are symmetric. Deadlock can arise if two processes both unwittingly specify only role $recipient_1$ even though role $recipient_2$ would equally suffice. To avoid this dilemma, a process attempting to enrole as a recipient would, somewhat unnaturally, target all recipient roles as follows:

$$
\begin{array}{ll}
[& \textbf{enrole } \textit{recipient}_1(x)@\textit{bi-cast} \longrightarrow \textbf{skip} \\
\Box & \textbf{enrole } \textit{recipient}_2(x)@\textit{bi-cast} \longrightarrow \textbf{skip} \;]
\end{array}
$$

To facilitate a more pleasant form of enrolement, symmetric roles can be replaced by a single *generic role* with a constant indicating the number of instances of the role to be filled. A process can then target this generic role without concern as to which instance it will actually fill. The constant could even be parameterized and supplied at run time by another enroler. Consider again our multicast example. We can define a first-order interaction with a transmitter and a generic recipient role which can be filled by m processes as follows:

$$
\begin{array}{l}
\textbf{interaction } \textit{m-cast} :: \\
[\quad \textbf{role } \textit{transmitter} \; (\textbf{value } m, z \text{: } \textbf{integer}) :: \textbf{skip} \\
\quad\;\; \textbf{role } \textit{recipient} \; [m] \; (\textbf{result } u \text{: } \textbf{integer}) :: u := z \;\;]
\end{array}
$$

Here the transmitter would supply the actual value of m, and, as a result, instances of the interaction can be activated with somewhat different role structures.

Within the setting of generic roles, one can reasonably restrict a process to be ready for at most one role of an interaction at a time without increasing the likelihood of deadlock.

[6] A *matching* M of an undirected graph $G = (V, E)$ is a subset of E such that no two edges of M share the same node. M is *maximum* if $|M'| \leq |M|$ for any matching M' of G. Given an undirected graph $G = (V, E)$, the *Maximum Matching* problem is to find a maximum matching of G.

6. EXTENSIONS OF THE EG ALGORITHM

6.1 Dynamically Changing Environments

Our algorithm can be extended to a dynamically changing environment where processes are created and destroyed at run time. We assume a model in which at any time processes can spawn new processes. However, a process can be destroyed, by itself or by its parent, only when it is not in an enrolement phase. Furthermore, when a process P_j creates a new child, say P_{jc}, that is capable of playing role r of I, P_j informs each existing process $P_i \in \mathcal{P}(I)$ of the identity of the new process so that P_i (and $Coord_i$, of course) can update its copy of $\mathcal{P}(r)$. (Recall that $\mathcal{P}(I)$ and $\mathcal{P}(r)$ are initially determined at compile time and each process in $\mathcal{P}(I)$ maintains a local copy.) In return, P_i sends P_j an acknowledgement message having timestamp t_i. After receiving the acknowledgements, P_j sets the logical clock of P_{jc} to be larger than all the t_i's.

While waiting for its parent to set its clock, P_{jc} "defers" any capture requests that may arrive. These deferred requests will be rejected by P_{jc} when its clock is finally set. Similarly, when a process $P_j \in \mathcal{P}(I)$ is destroyed, it informs the existing processes in $\mathcal{P}(I)$ to remove P_j from their local copy of $\mathcal{P}(I)$.

Care must be taken to insure that *concurrently created processes* keep their local copies of $\mathcal{P}(r)$ up-to-date. Consider, for example, the scenario in which processes P_i and P_j are about to inform each other about the birth of their child processes P_{ic} and P_{jc}, respectively. Assume that P_{ic} and P_{jc} are both capable of playing role r, the role in question. If P_i informs P_j first (i.e., P_j receives P_i's message about P_{ic} before P_j sends its message about P_{jc} to P_i), then P_j must also inform P_{ic} about the birth of P_{jc}. Furthermore, P_j must assume the responsibility of updating P_{jc}'s copy of $\mathcal{P}(r)$. If P_i and P_j inform each other simultaneously (i.e., P_i sends its message about P_{ic} to P_j, but before the message is acknowledged, P_i receives P_j's message about P_{jc}, and similarly for P_j), then both P_i and P_j must update P_{ic}'s and P_{jc}'s copy of $\mathcal{P}(r)$, respectively.

Since a process is not destroyed when it is in an enrolement phase, the safety property of Theorem 1 is still guaranteed. The liveness property holds as well because a newly created process P_j is always younger than any existing coordinator $Coord_i$ that is building a quorum for some interaction. Hence, when $Coord_j$ is activated by P_j, $Coord_j$ cannot interfere with $Coord_i$ as $Coord_j$ is unable to contend with $Coord_i$ in capturing a process (see Section 3). Also, $Coord_i$ need not attempt to capture P_j as such an attempt is doomed to fail anyway since a coordinator cannot capture a younger process. Therefore, $Coord_i$ need only request capture of those processes created before P_i entered its current enrolement phase. Fortunately, this set of processes is finite.

6.2 Interaction Scheduling in IP

As remarked in Section 1, enrolement into a team in IP does not impose any synchronization delays on a process, but rather the interactions to be executed first within the role. To illustrate, consider the following team taken from Francez and Forman [1990a]:

team T ::
[**role** R_1 (**value** n_1: **integer**) ::
  [B_1 & $a[\,] \longrightarrow \ldots$

$\quad\quad\quad\quad\square\ C_1\ \&\ e[\,]\longrightarrow \ldots\,]$
$\|\ \ \textbf{role}\ \ R_2\ (\textbf{value}\ n_2\text{: }\textbf{integer})::$
$\quad\quad\quad\quad[\ \ B_2\ \&\ a[\,]\longrightarrow \ldots$
$\quad\quad\quad\quad\square\ C_2\ \&\ e[\,]\longrightarrow \ldots\,]$
$\|\ \ \textbf{role}\ \ R_3\ (\textbf{value}\ n_3\text{: }\textbf{integer})::$
$\quad\quad\quad\quad[\ \ B_3\ \&\ a[\,]\longrightarrow \ldots$
$\quad\quad\quad\quad\square\ C_3\ \&\ e[\,]\longrightarrow \ldots\,]$
$]$

Interactions a and e each have as their participants three formal processes, roles R_1, R_2, and R_3, and as such are "pure" first-order interactions. A process P can execute the enrolement statement $R_1(x)@t$ as a *local* action, i.e. without being delayed, to enrole into role R_1 of an instance of team T. After enroling into R_1, P is now faced with an alternative command where it is ready to participate in interactions a or e. Our EG algorithm can then be used to coordinate an interaction for P.

The situation is somewhat different in IP if an enrolement statement is to be executed as a guard to an alternative command. In this case, the decision to execute the guard is propagated across team enrolement to the set of *first actions* within the role targeted by the enroler, a process known as *conflict propagation* [Francez and Forman 1990a]. Thus, although enrolement guards do not delay a process in any semantic sense, conflict propagation makes it look like they *do* delay a process. For example, consider again the above team T. Assume that P would like to execute the alternative command

$$\begin{array}{ll}[\ R_1(x)@t & \longrightarrow S_1\\ \square\ b[\ldots] & \longrightarrow S_2\ ]\end{array}$$

to enrole into role R_1 or to participate in an interaction b. The set of first actions of R_1 consists of interactions a and e. Only if P has committed to participate in a or e can P execute the enrolement guard $R_1(x)@t$. Thus, to execute the alternative command, P must first decide in which interaction, a, e, or b, to participate. Our EG algorithm can be used at this point to coordinate an interaction for P.

The above two examples illustrate how our EG algorithm can be used to implement enrolement statements (or, schedule first-order interactions) in IP. Note that because enrolement statements take on different semantics depending upon whether or not they serve as guards, in the first example, the EG algorithm is invoked *after* a process has enroled into a team, while in the second example, the algorithm is invoked *before* the process has enroled into a team.

6.3 Bi-Party First-Order Interaction

The EG algorithm can be used to schedule bi-party first-order interactions such as those found in CCS [Milner 1989]. As described in Section 3.2, the EG algorithm can be fine-tuned such that when a coordinator's capture request has been deferred, the coordinator can either proceed conservatively by waiting for its request to be granted or rejected, or aggressively by sending a new capture request to another yet-to-be-requested process. In the multiparty case, both approaches yield the same message complexity: at most $6m$ messages can be generated when a process is building a quorum of I, where $m = |\mathcal{P}(I)|$ is the number of processes that can potentially enrole into I. In the bi-party case, however, the two approaches yield different message complexity: the conservative approach generates at most $2m$ messages, and the aggressive approach generates at most $4m$ messages. Note that the message complexity goes down from the multiparty to the bi-party case. This reduction in message complexity has also been observed for zeroth-order interactions [Bagrodia 1989b].

6.4 Nested Enrolement

So far, we have not addressed the possibility of allowing enrolement statements within roles. This leads to a notion of *nested enrolement*, which exists in some form already in IP and Script. The EG algorithm can be safely applied at each level of enrolement: level n enrolement will involve roles of level $n-1$, which are eventually filled by actual processes, resulting in an instance of unnested first-order interaction enrolement. Note that as pointed out by Francez and Forman [1990a] the chain of nested enrolement should be kept short to reduce the amount of work needed for synchronization.

6.5 Partners-Named Enrolement

Francez, Hailpern, and Taubenfeld [1986] propose a *partners-named* enrolement scheme that allows a process to name some or all of the other enrolers. For a set of processes to activate an interaction, their naming specifications must be consistent in the sense that if P_i names P_j to fill role r_k, then P_j indeed enroles into r_k. Such a scheme is useful, for example, if a process P wants to multicast to a particular group G of recipients. To ensure that the message is received only by processes in G, P names exactly these processes as recipients.

We show in [Joung 1992], however, that the enrolement guard scheduling problem for partners-named enrolement is NP-complete, even when enrolement statements cannot act as guards. We additionally present two naming conventions, both of which permit enrolement guards, for which the problem can be solved efficiently by the EG algorithm. The first requires a process to name either *all* of the participants of an interaction or *none* of them, while the second allows a process to name only those participants with which it directly exchanges data (but it cannot name the same process for the same role in more than one guard of an alternative command). To compare, the first naming convention is simpler to formulate, but the second may be more useful in practice.

7. CONCLUSIONS

We have presented a fully distributed and message-efficient algorithm for the enrolement guard scheduling problem, the first such algorithm of which we are aware. We have also described several extensions of the algorithm, including generic roles, dynamically changing environments, interaction scheduling in IP, bi-party interactions, and nested enrolement.

Recall that the participants of a multiparty interaction are required to synchronize in order to establish the interaction. So far we have not discussed whether or not the processes in an interaction are required to synchronize to terminate the interaction, a phenomenon known as *exit synchronization*. It is observed by Evangelist, Francez, and Katz [1989] and Katz, Forman, and Evangelist [1990] that exit synchronization is not necessary if multiparty interaction is the only source of synchronization among processes.

Various notions of fairness for multiparty interactions have been proposed in the literature [Francez 1986; Apt, Francez, and Katz 1988; Attie, Francez, and Grumberg 1990; Attie, Forman, and Levy 1990], including: *weak interaction fairness*, where an interaction that is continually enabled[7] will eventually be disabled, and *strong interaction fairness*, where an interaction that is infinitely often enabled will be established infinitely often. The liveness property of our EG algorithm (Theorem 2) implies that the algorithm is weak interaction fair. It is not, however, strong interaction fair. In fact, it is shown in [Tsay and Bagrodia 1993; Joung 1992] that no message-passing-based algorithm can be strong interaction fair if message transmission delays are non-negligible.

ACKNOWLEDGMENTS

We would like to thank the anonymous referees for their invaluable comments, which significantly helped to improve the quality of the paper. We are also grateful to Nissim Francez for providing us with several references on IP and important insights into the language, and Shaji Bhaskar for his suggestions on how to present the EG algorithm.

References

Apt, K., Francez, N., and Katz, S. 1988. Appraising fairness in languages for distributed programming. *Distributed Computing 2,* 4, 226–241. Also: *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, Munich, Germany, 1987.

Attie, P., Forman, I., and Levy, E. 1990. On fairness as an abstraction for the design of distributed systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, pp. 150–157.

Attie, P., Francez, N., and Grumberg, O. 1990. Fairness and hyperfairness in multi-party interactions. In *Proceedings of the 17th ACM Annual Symposium on Principles of Programming Languages*, San Francisco, California, pp. 292–305. ACM Press.

Back, R. and Kurki-Suonio, R. 1988. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems 10,* 4 (Oct.), 513–544.

Bagrodia, R. 1989a. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering SE-15,* 9 (Sept.), 1053–1065.

Bagrodia, R. 1989b. Synchronization of asynchronous processes in CSP. *ACM Transactions on Programming Languages and Systems 11,* 4 (Oct.), 585–597.

Chandy, K. and Misra, J. 1988. *A Foundation of Parallel Program Design.* Addison-Wesley.

Charlesworth, A. 1987. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems 9,* 2 (July), 350–366.

Dijkstra, E. 1965. Solution of a problem in concurrent programming control. *Communications of the ACM 8,* 9 (Sept.), 569.

[7]An interaction is *enabled* if all of its participants are ready for the interaction, and is *disabled* otherwise.

EVANGELIST, M., FRANCEZ, N., AND KATZ, S. 1989. Multiparty interactions for interprocess communication and synchronization. *IEEE Transactions on Software Engineering SE-15,* 11 (Nov.), 1417–1426.

FORMAN, I. 1986. On the design of large distributed systems. In *Proceedings of the First International Conference on Computer Languages*, Miami, Florida, pp. 84–95.

FRANCEZ, N. 1986. *Fairness.* Springer-Verlag.

FRANCEZ, N. 1990. Private communication.

FRANCEZ, N. AND FORMAN, I. 1990a. Conflict propagation. In *Proceedings of the Third International Conference on Computer Languages*, New Orleans, Louisiana, pp. 155–168.

FRANCEZ, N. AND FORMAN, I. 1990b. Interacting Processes: Coordinated distributed programming. Technical report stp-271-88 (July), Microelectronics and Computer Technology Corp., Austin, Texas. A preliminary version presented in the 5th Jerusalem Conference on Information Technology.

FRANCEZ, N. AND FORMAN, I. 1994. *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming.* Addison Wesley.

FRANCEZ, N., HAILPERN, B., AND TAUBENFELD, G. 1986. Script: A communication abstraction mechanism. *Science of Computer Programming 6,* 1 (Jan.), 35–88.

HOARE, C. 1978. Communicating sequential processes. *Communications of the ACM 21,* 8 (Aug.), 666–677.

HOPCROFT, J. AND KARP, R. 1973. An $n^{5/2}$ algorithm for maximum matching in bipartite graph. *SIAM Journal on Computing 2,* 4 (Dec.), 225–231.

JOUNG, Y.-J. 1992. *On the Design and Implementation of Multiparty Interaction.* Ph. D. thesis, Department of Computer Science, State University of New York at Stony Brook, New York.

JOUNG, Y.-J. AND SMOLKA, S. 1990. Efficient, dynamically structured multiprocess communication. In *Proceedings of the 28th Annual Allerton Conference on Communication, Control, and Computing.*

JOUNG, Y.-J. AND SMOLKA, S. 1992. A comprehensive study of the complexity of multiparty interaction (extended abstract). In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, pp. 142–153. ACM Press.

KATZ, S., FORMAN, I., AND EVANGELIST, M. 1990. Language constructs for distributed systems. In *IFIP TC2 Proceedings of Working Conference on Programming Concepts and Methods*, Sea of Galilee, Israel, pp. 70–97.

KUMAR, D. 1990. An implementation of N-party synchronization using tokens. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, pp. 320–327.

LAMPORT, L. 1978. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21,* 7 (July), 558–565.

MILNER, R. 1989. *Communication and Concurrency.* International Series in Computer Science. United Kingdom: Prentice Hall.

PARK, M. AND KIM, M. 1990. A distributed synchronization scheme for fair multi-process handshakes. *Information Processing Letters 34*, 131–138.

RAMESH, S. 1987. A new and efficient implementation of multiprocess synchronization. In *Proceedings Conference on PARLE, Lecture Notes in Computer Science 259*, Berlin, pp. 387–401. Springer-Verlag.

RAMESH, S. AND MEHNDIRATTA, S. 1985. A new class of high-level programs for distributed computing systems. In *Proceedings of the Fifth Conference on FST-TCS, Lecture Notes in Computer Science 206*, Berlin, pp. 42–72. Springer-Verlag.

TSAY, Y.-K. AND BAGRODIA, R. 1993. Some impossibility results in interprocess synchronization. *Distributed Computing 6,* 4, 221–231.

U.S. DEPARTMENT OF DEFENSE 1983. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A. Washington, D.C.: U.S. Government Printing Office.