Int. Computer Symposium, Dec. 15-17, 2004, Taipei, Taiwan.

Coordination Support for Heterogeneous Distributed Applications

Chia-Chu Chiang
Department of Computer Science
University of Arkansas at Little Rock
2801 S. University Ave., Little Rock, AR72204-1099, USA
E-mail: cxchiang@ualr.edu

Abstract - *We present an approach to supporting the development of heterogeneous distributed applications for coordination through Multiparty Interaction (MI) protocol. A CORBA middleware technology is used as an underlying communication infrastructure to support heterogeneous communications. The approach decouples the applications and their underlying middleware implementations including coordination protocols by providing a set of generic interfaces to the applications.*

Keywords: Coordination, CORBA, Middleware, Multiparty Interaction

1. Introduction

To support the development of heterogeneous distributed applications for coordination, we augment the existing middleware technologies to provide collaboration support through Multiparty Interaction (MI) protocol. An approach is presented to decouple the applications and their underlying middleware implementations including coordination protocols by providing a set of generic interfaces to the applications.

Joung and Smolka [4] writes that "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." N. Francez and I. R. Forman [3] present IP (Interacting Process) as the basis of specification languages for multiparty interaction. In IP, a distributed system is organized into teams. A team is viewed as a collection of distributed processes that interact with each other through multiparty interactions. A process can participate in a multiparty interaction through an interaction statement of the form a[…], where *a* is the interaction name and […] includes the statements to be executed by this process when the interaction point is executed.

In this research, we are not implementing a new language processor to execute IP specifications. On the contrary, our intention is to allow IP specifications to be realized under any general programming environment. The approach we use is to analyze an IP specification and generate a multiparty interaction description that is a data structure to describe the properties of the multiparty interactions in IP. Our IP language mapping approach allows a multiparty interaction description written in any target programming language to be automatically generated from an IP specification. Application developers then write a program in the target language to include the multiparty interaction description in the program. A function in the coordination library will be invoked to represent the caller (participating party) to interact with other participants for coordination. In this paper, we are focusing on the coordination support in heterogeneous distributed programming.

2. Coordination support

Basically, the implementation of our distributed multiparty interactions consists of three phases: synchronization, data exchange, and computation. In the synchronization phase, enabled interactions are detected and one is selected for execution. In the data exchange phase, data are exchanged among participating processes through the underlying middleware. In the computation phase, upon receiving all the needed data, the processes participating in an interaction continue their executions on the interaction bodies. For example, in the dining philosophers problem shown in Figure 1, Philosopher$_0$, Fork$_0$, and Fork$_3$ need to synchronize at the interaction point get_fork$_0$ in the synchronization phase. Next, they start to exchange data in the data exchange phase. In this problem, however, there is no need for data exchange among Philosopher$_0$, Fork$_0$, and Fork$_3$. Following the data exchange phase, these three participants enter the computation phase, Philosopher$_0$ needs to execute the body of get_fork$_0$ by assigning *'eating'* to s_0 which is a local variable declared in the Philosopher$_0$ process.

```
DINING_PHILOSOPHERS :: [Philosopher$_0$ || Philosopher$_1$ || Philosopher$_2$ || Philosopher$_3$ || Fork$_0$ || Fork$_1$ || Fork$_2$ || Fork$_3$], where

Philosopher$_i$ :: i = 0, 3
$s_i$ := 'thinking';
*[$s_i$ = 'thinking' $\rightarrow$ $s_i$ := 'hungry'
  □
  $s_i$ = 'hungry' & get_fork$_i$[$s_i$ := 'eating'] $\rightarrow$ release_fork$_i$[]
 ]
```

```
Fork$_i$ :: i = 0, 3
*[get_fork$_i$[] $\rightarrow$ release_fork$_i$[]
  □
  get_fork$_{(i+1) \bmod 4}$[] $\rightarrow$ release_fork$_{(i+1) \bmod 4}$[]
 ]
```

Figure 1. An IP to the dining philosophers

Centralized solutions to the implementation of multiparty interactions work quite well [2]. Our solution shifts the centralized solutions to a distributed solution. The work to be done in the three phases is distributed among participants and their thread managers. Each participating process creates its own thread manager to mange its interactions. For an interaction, which the participating process gets involved in, the thread manager will create a proxy thread to connect to the interaction. During the synchronization phase, information about enablement or disablement of interactions is exchanged. Once one interaction is selected, the thread manager notifies the other threads within the participating process indicating their interactions are not selected, so no one will be neglected and the whole process is fair. Figure 2 depicts our solution in the case of four philosophers who are trying to pick up their forks. The detailed implementation of coordination support is described in Section 3.

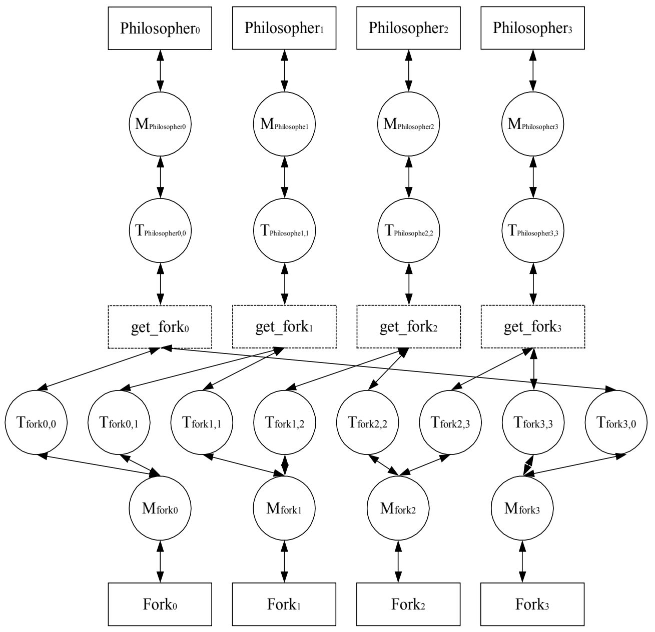

Figure 2. A distributed solution to the dining philosophers problem

After synchronization, data exchange takes place. Thread managers inform their participating processes which interactions have been selected for execution. The participating processes exchange the data they are responsible for with their corresponding thread managers by means of *PutData(INOUT &OperationArgumentBuffer)* and *GetData(INOUT, &OperationArgumentBuffer). GetData()* and *PutData()* transmit data through the invocation of CORBA functions. At this moment, no participating processes can continue until they all have completed data exchanges. The detailed descriptions of the coordination support can be found in [1].

3. Implementation of the coordination support

The heart of the design and implementation of multiparty interactions is the distributed guard scheduling problem described as follows:

Given n multiparty interactions I_i (i=1,…, n), each of which has l_i parties to be participated by distinct processes form m participating processes P_j (j=1,…,m) whose identifiers are not know until run-time, the guard scheduling problem is to select at subset of the multiparty interactions for execution, subject to the following constraints,

1. Each interaction selected for execution must have all its parties participated by distinct processes.
2. No process can participate in executions of more than on interaction.
3. If there are interactions which can be selected for execution, the selection must be finished in finite time.

Constraints 1 and 2 above are the safety requirement and Constraint 3 the liveness requirement of the problem.

For each interaction I_i, we create an interaction process also denoted I_i. If P_j is ready to participate in k interactions I_{i1}, ..., I_{ik}, we create (1) thread manager M_j and (2) one proxy thread $T_{j,ir}$ for each of I_{i1}, ..., I_{ik}. Our algorithm consists of three protocols for proxy thread $T_{j,ir}$, thread manager M_j, and interaction process I_{ir} . Each of the protocols is a finite state machine which uses input message as event for state transition. The proxy thread $T_{j,ir}$ is used to communicate with interaction process, I_{ir}. Thread manager M_j serves as the manager of all the proxy threads $T_{j,i1}, \ldots T_{j,ik}$ which it spawns.

The basic idea of the protocol for $T_{j,ir}$ is as follows: It sends message *Request* to I_{ir} to notify its intension to participate. When I_{ir} receives all the *Requests* needed, it sends back a message *All-Met* to $T_{j,ir}$, telling $T_{j,ir}$ that I_{ir} is ready to be activated. After receiving message *All-Met*, $T_{j,ir}$ may do one of the following three tasks: (1) if none of the other $T_{j,ir'}$ has committed, $T_{j,ir}$ may proceed to commit itself to I_{ir} by sending a *Commit* message to it and makes transition to "commit" state (2) if a $T_{j,ir'}$ with higher priority has committed to $I_{ir'}$, $T_{j,ir}$ withdraw its participation by sending a *Withdraw* message to I_{ir} and makes transition to "re-try" state, or (3) if a $T_{j,ir'}$ with lower priority has committed to $I_{ir'}$, $T_{j,ir}$ makes transition to "pending" state waiting to commit in case the commitment of $T_{j,ir'}$ does not realize the actual activation of $I_{ir'}$ (due to withdrawals of other participants of $I_{ir'}$). The information about the commitment or pending of all proxy threads is stored in a shared array accessed through critical sections. Once in "commit" state, $T_{j,ir}$ is waiting for *Succeed* message from I_{ir} when it receives commitment from all of its participants. After $T_{j,ir}$ receives *Succeed* message, it sends a *Finish* message to thread manager M_j to register the activation of I_{ir} and make transition to "success" state. The state diagram of the protocol of thread $T_{j,ir}$ is shown in Figure 3. The transitions are represented by arcs labeled with event/action pairs.

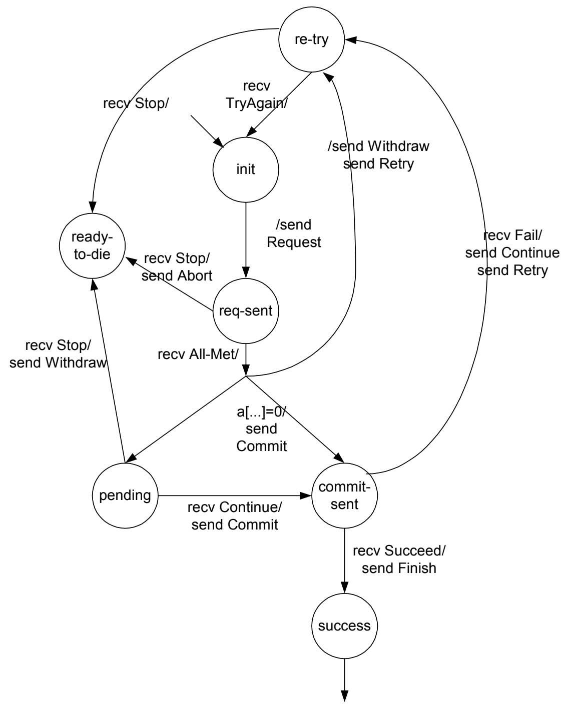

Figure 3. State diagram of thread $T_{j,ir}$

The protocol for interaction process, I_{ir}, is a simple two-phase locking protocol with three states: "meeting", "all-met" and "success". In the "meeting" state, I_{ir} receives *Request* message or *Abort* message from its participants, incrementing or decrementing its request counter, respectively. When the request counter reaches the total number of participants, I_{ir} sends *All-Met* message to all of the participants, and makes transition to "all-met" state. In the "all-met" state, it waits for either a *Commit*, *Withdrawal* or *Abort* message from each of its participants. A commit counter and a withdrawal counter are used to track the numbers of the corresponding participants to decide whether it can transit to "success" state (when all participants committed) or "meeting" state to start over again for the next round of coordination (when all responded, but the number of committed falls short of the total number of participants). The state diagram of I_{ir} is shown in Figure 4. Process I_{ir} maintains three counters, *nReq*, *nC*, and *nW*, for the number of Request()s, Commit()s, and Withdraw()s received, respectively.

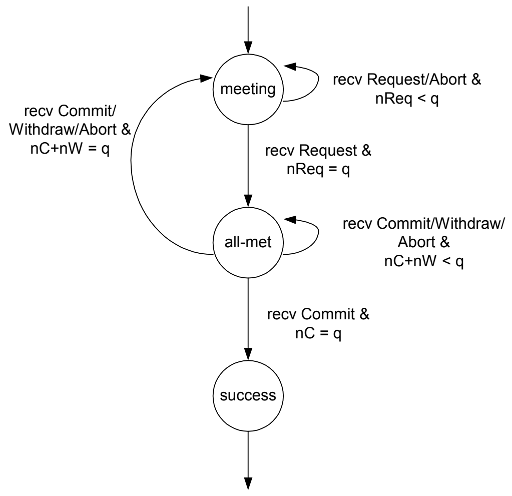
meeting
recv Request/Abort & nReq < q
recv Commit/ Withdraw/Abort & nC+nW = q
recv Request & nReq = q
all-met
recv Commit/Withdraw/ Abort & nC+nW < q
recv Commit & nC = q
success

Figure 4. State diagram of interaction process I_{ir}

The protocol for thread manager M_j is to coordinate its all the proxy threads. It also intercepts and relay messages between proxy threads and its corresponding interaction process. In particular, it will discard all the messages to $T_{j,ir}$ after it is killed by M_j. The main function of M_j is to synchronize the transitions of $T_{j,i1}, \ldots T_{j,ik}$. After it spawns the proxy threads $T_{j,i1}, \ldots, T_{j,ik}$, it waits for either *Re-try* message or *Finish* message from each of them. Upon receiving a *Finish* message from $T_{j,ir}$, it sends *Stop* message to all the other proxy threads so that they can send *Withdrawal* or *Abort* message to its corresponding interaction manages before make transition to 'ready-to-die" state. If all proxy threads send *Re-try* message, M_j sends "start-over" message back and allow them to start the next round of coordination. The state diagram of thread M_j is shown in Figure 5. Thread M_j maintains a counter, *nRetry*, to synchronize all the proxy threads before entering into the next round of coordination. The next round of coordination should not start until all the k proxy threads $T_{j,ir}$ (r = 1, …, k) fail.

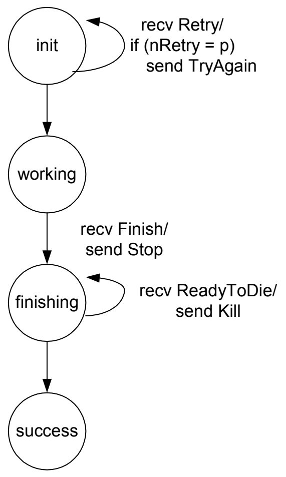
init
recv Retry/ if (nRetry = p) send TryAgain
working
recv Finish/ send Stop
finishing
recv ReadyToDie/ send Kill
success

Figure 5. State diagram of thread M_j

4. Correctness and complexity

In this section, we prove the correctness and analyze the message complexity of the guard scheduling algorithm presented in the previous section. A solution to the guard scheduling problem for coordinating first-order multiparty interactions must satisfy the requirements of safety, liveness, and progress.

4.1 Safety

The safety requirement of the guard scheduling problem defined in Section 3.3 demands that

- no interaction be selected to execute unless all its parties are participated by distinct processes (interaction safety), and
- no process participates in more than one interaction at a time (process safety).

The interaction safety requirement can be derived from the protocol of I_{ir} directly. In particular, process I_{ir} will not enter into state 'all-met' unless it receives the requests for participation from q (i.e. l_{ir}) processes. Furthermore, it will not enter into state 'success' unless it receives the commitments from all these processes.

The process safety is ensured by Theorem 1 as follows.

Theorem 1. Among the proxy threads $T_{j,i1}$, …, $T_{j,ik}$, started by P_j, only one can enter into state 'success'.

Proof: Thread $T_{j,ir}$ can enter into state 'success' only from state 'commit-sent'. It can enter into state 'commit-sent' either from state 'pending' or state 'request-sent'. Thread $T_{j,ir}$ moves from state 'request-sent' to state 'commit-sent' only when all the bits of bit map a[] are 0. If it moves into state 'pending', it will not enter into state 'commit-sent' until another thread sends it a *Continue* after leaving state 'commit-sent'. Therefore, there is only one thread in state 'commit-sent' at any time. After the thread enters into state 'success', all other threads will be killed.

4.2 Liveness

The liveness requirement of the guard scheduling problem demands that there be no deadlock in the system comprising all the threads and processes running the protocols of $T_{j,ir}$, M_j, and I_{ir}. In particular, no process or thread is allowed to stay in a waiting state indefinitely.

After I_{ir} receives all the I_{ir} requests it needs and enters into state 'all-met', it will receive the same number (l_i) of Commit(), Withdraw() or Abort(), provided that each thread $T_{j,ir}$ involved is live and responds eventually. After that, I_{ir} will enter either into state 'meeting' again for the next round of coordination or into state 'success'. In other words, I_{ir} is live as long as each thread $T_{j,ir}$ with which it communicates is live.

Similarly, if every thread $T_{j,ir}$ ($1 \le r \le k$) is live, thread M_j is also live. In particular, thread M_j will remain in state 'working' and start the next round of coordination if all the k proxy threads $T_{j,ir}$ (r = 1, …, k) are successful. If one thread succeeds, M_j will receive Finish() from it and enter into state 'finishing'. M_j will further receive (k-1) ReadyToDie()s from the remaining proxy threads and enters into state 'success'. Therefore, the liveness of the entire system hinges on the liveness of the protocol of $T_{j,ir}$. The following lemma is used to prove the liveness of $T_{j,ir}$.

Lemma 1. If a proxy thread $T_{j,ir}$ is in state 'pending' indefinitely, there must be another proxy thread $T_{j,ir'}$ of P_j such that r < r' in state 'commit-sent' indefinitely.

Proof: a[r] = 1 only if $T_{j,ir}$ is in state 'commit-sent' or 'pending', but the first thread $T_{j,ir}$ with a[r] = 1 must be in state 'commit-sent'. To simplify the notation, we rename $T_{j,ir}$ to be $T'_{j,r}$. Let us assume that $T'_{j,r}$ stays in state 'pending' indefinitely.

When thread $T_{j,ir}$ enters into state 'pending', $a[(r+1)\ldots k] \neq 0$ must be held. Let $a[u_1]$, …, $a[u_v]$ ($r+1 \le u_1 < \ldots < u_v \le k$) be all the bits that either are 1 when $T'_{j,r}$ enters into state 'pending' or ever become 1 when $T'_{j,r}$ is in state 'pending' (indefinitely).

Thread $T'_{j,uv}$ must be in state 'commit-sent' when $T'_{j,r}$ enters into state 'pending'. Other threads $T'_{j,u1}$, …, $T'_{j,uv-1}$ must be in state 'pending' first. We want to prove that based on the assumption above at least one of $T'_{j,u1}$, …, $T'_{j,uv}$ must be in state 'commit-sent' indefinitely.

Consider thread $T'_{j,uv}$ first. If it does not stay in state 'commit-sent' indefinitely, it must receive a Success() or a *Fail* in finite time. If it receives a Success(), $T'_{j,r}$ would leave state 'pending' in finite time. This would contradict the assumption above. If it receives a *Fail*, thread $T'_{j,uv-1}$ will enter into state 'commit-sent' in finite time. The same procedure will also apply to threads $T'_{j,uv-1}$, …, $T'_{j,u1}$. Therefore, if none of $T'_{j,u1}$, …, $T'_{j,uv}$ can stay in state 'commit-sent' indefinitely, $T'_{j,r}$ will leave state 'pending' in finite time. This proves the lemma.

There are four waiting states in the protocol of $T_{j,ir}$: 'req-sent', 'commit-sent', 'pending', and 're-try'. The waiting of $T_{j,ir}$ in state 'req-sent' is to ensure interaction safety and should not be considered as a problem for liveness. $T_{j,ir}$ in state 're-try' will enter into state 'init' after all the threads started by P_j send Withdraw()s to their interactions. Therefore, for the liveness of the protocol of $T_{j,ir}$, we only need to prove that no thread $T_{j,ir}$ will stay in state 'commit-sent' or 'pending' indefinitely. This is done in the following theorem.

Theorem 2. It is impossible for any proxy thread $T_{j,ir}$ in the system to stay in state 'commit-sent' or 'pending' indefinitely.

Proof: According to Lemma 1, we only need to prove that it is impossible for any proxy thread $T_{j,ir}$ to stay in state 'commit-sent' indefinitely.

Let us assume that there is a proxy thread $T_{j1,i1}$ staying in state 'commit-sent' indefinitely. This means that $T_{j1,i1}$

receives neither Success() nor *Fail* in finite time. Therefore, none of the threads coordinating interaction I_{i1} ever sends a Withdraw() or an Abort() to it. Furthermore, there is at least one of these threads that does not ever send a Commit() either. Let this thread be $T_{j2,i1}$. According to the protocol, $T_{j2,i2}$ from the same process P_{j2} such that it stays in state 'commit-sent' indefinitely and $i_1 < i_2$. Continuing this way, we will have an infinite series

$$T_{j1,i1}, T_{j2,i1}, T_{j2,i2}, \ldots, T_{jk,ik-1}, T_{jk,ik}, \ldots$$

such that $T_{jk,ik}$ ($1 \leq k$) and $T_{jk,ik-1}$ ($2 \leq k$) are indefinitely in states 'commit-sent' and 'pending', respectively, and $i_1 < i_2 < \ldots < i_k < \ldots$. On the other hand, there are only a finite number (m) of interactions and we must have $i_1 < i_2 < \ldots < i_k < \ldots < m$. Therefore, the series above cannot be infinite. We have reached a contradiction.

4.3 Progress

We have proved the liveness of the system. The next question is whether the system can make progress in selecting interactions. The liveness of the system guarantees that an interaction process in state 'all-met' will enter into state 'meeting' or state 'success' in finite time. The progress requirement demands that at least one of those interactions in state 'all-met' enter into state 'success'. This requirement is satisfied in our algorithm. In order to prove this, we need the lemma as follows.

Lemma 2. If thread $T_{j,ir}$ sends Withdraw() to I_{ir} in state 'req-sent' and enters into state 're-try', there must be another thread $T_{j,ir'}$ of P_j in state 'commit-sent' such that $r' < r$.

Proof: Thread $T_{j,ir}$ in state 'req-sent' sends a Withdraw() to I_{ir} only if it finds $a[1..(r-1)] \neq 0$. Let r' be the largest integer such that $r' < r$ and $a[r'] = 1$. According to the protocol, thread $T_{j,ir'}$ is either the first thread in state 'commit-sent' or a past pending thread which has been woken up by another thread and entered into state 'commit-sent'.

The following theorem shows that in each coordination at least one selectable interaction will be selected. This ensures the progress of our algorithm.

Theorem 3. Let $I_{u1}, \ldots, I_{uw}$ be the subset of all the interactions that enter into state 'all-met' after receiving all the requests they need. Then, at least one of them will enter into state 'success'.

Proof: Let $P_{v1}, \ldots, P_{vy}$ be all the processes involved to make $I_{u1}, \ldots, I_{uw}$ enter into state 'all-met'. Without loss of generality, we also assume $I_{u1} < \ldots < I_{uw}$, i.e. $u1 < \ldots < uw$. Due to the liveness of the system, every interaction of $I_{u1}, \ldots, I_{uw}$ will receive a response, Commit(), Withdraw(), or Abort(), from each of its participating processes from $P_{v1}, \ldots, P_{vy}$ and enter into either state 'meeting' or state 'success'. Let us assume that none of $I_{u1}, \ldots, I_{uw}$ enters into state 'success'.

According to the protocol, a thread $T_{vj,ui}$ ($1 \leq j \leq y$, $1 \leq i \leq w$) can send Withdraw() only in two states: 'req-sent' and 'pending'. But, based on the assumption above, it is impossible for $T_{vj,ui}$ to send Withdraw() in state 'pending'. This is because otherwise it must receive a *Stop* from M_{vj} and therefore there must be a thread $T_{vj,ui'}$ ($1 \leq i' \leq w$) that succeeds in its coordination. This would imply that $I_{ui'}$ enters into state 'success'.

To simplify the notation, P_{vj}, I_{ui}, and $T_{vj,ui}$ are renamed P'_j, I'_i, and $T'_{j,i}$, respectively. Consider I'_w first. Because it enters into state 'meeting', it must have received at least one Withdraw() from, say T'_{j1},w ($1 \leq j_1 \leq y$). According to Lemma 2, there must be a thread $T'_{j1,i1}$ that has sent a Commit() to I'_{i1} such that $i_1 < w$. Since I'_{i1} also enters into state 'meeting', it must have received a Withdraw() from say, $T'_{j2,i1}$ ($1 \leq j_2 \leq y$). By using Lemma 2 again, we can find another thread $T'_{j2,i2}$ that has sent a Commit() to I'_{i2} such that $i_2 < i_1$. Continuing this way, we will have an infinite series

$$T'_{j1,w}, T'_{j1,i1}, T'_{j2,i1}, \ldots, T'_{jk,ik}, T'_{jk+1,ik}, \ldots$$

such that $\ldots < i_k < \ldots < i_1 < w$. On the other hand, there are only a finite number (w) of interactions involved and we must have $1 < \ldots < i_k < \ldots < i_1 < w$. Therefore, the above series cannot be infinite. We have reached a contradiction.

4.4 Message complexity

In our algorithm, I_{ir} will re-try in the next round of coordination if it is not selected. Theorem 3 shows that at least one selectable interaction will be selected in each round of coordination. For a particular interaction selected eventually, the cost is obviously the number of rounds of coordination it has gone through times the number of messages required in each round of coordination. According to the protocols of our algorithm, $4l_{ir}$ messages between interaction I_{ir} and its l_{ir} participating processes are required in each round of coordination.

The average number of rounds of coordination required to select an interaction depends on many factors. First of all, it depends on the number of interactions connected through processes in conflict (processes ready to participate in more than one interactions) in the bipartite graph [1]. The larger is this number, the more rounds of coordination are needed. Secondly, the larger is the average number of interactions in which processes in conflict participate, the faster drops the number of selectable interactions. As a result, the average number of rounds of coordination required to select selectable interactions would be smaller. The third factor is the non-deterministic nature of the algorithm. Figure 6 shows two possible scenarios of selection of interactions I_0 (P_0, F_0, and F_3) and I_2 (P_1, F_0, and F_1) in Figure 1. I_i represents the multiparty interactions, get_fork$_i$, of the fork processes Fork$_i$, Fork$_{(i-1) \bmod 4}$, and Philosopher$_i$ (we assume that index arithmetic is cyclic, i.e., $0 - 1 = 3$ and $3 + 1 = 0$).

Labels *C* or *W* of an edge shows that the process has sent Commit() or Withdraw(), respectively, to the corresponding interaction. Label *P* indicates that the process is in state 'pending' after it receives *All-Met* from the corresponding interaction. Label R indicates that the process has sent a Request() to the interaction, but cannot receive *All-Met* from it.

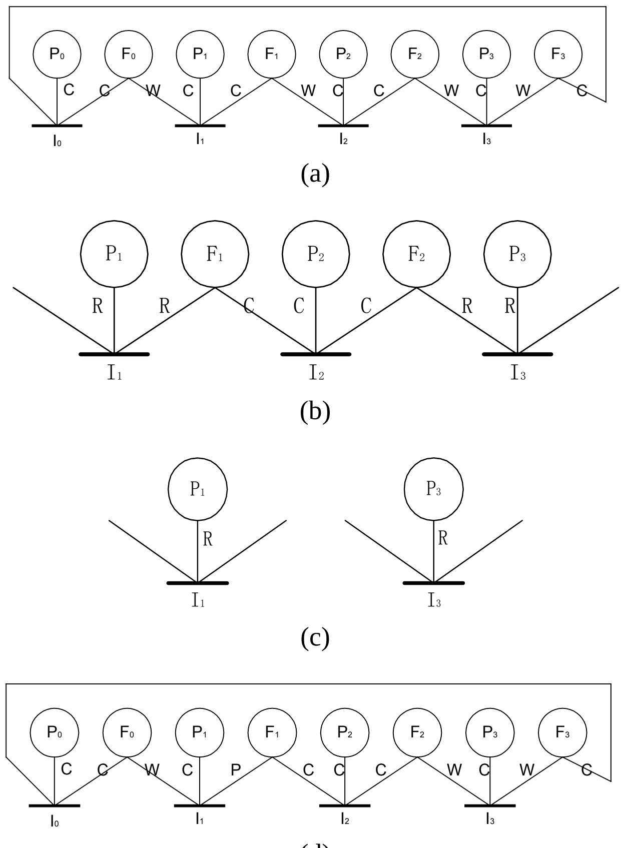

Figure 6. Progress of non-deterministic selection

Figure 6(a) shows one possible situation where only interaction I_0 is about be selected. The selection of I_0 leaves only I_2 still selectable in the second round of coordination. Figure 6(b) shows how I_2 is selected in the second round of coordination. Note that I_1 and I_3 cannot receive Request()s from F_0 and F_3 ($T_{F_1,1}$ and $T_{F_3,3}$, to be exact)., respectively, because they are not available anymore. Figure 6(c) shows the situation after I_2 is selected.

Let us go back to the situation shown in Figure 6(a) again. If the *All-Met* of I_2 reached thread $T_{F_1,2}$ of F_1 before the *All-Met* of I_1 reached thread $T_{F_1,1}$ of F_1, we would like have the situation shown in Figure 6(d). Both I_0 and I_2 would be selected in the first round of coordination and the system would move to the situation shown in Figure 6(c) immediately.

As a matter of a fact, the scenario shown above in Figure 6(a), (b), and (c), is the worst case that can ever happen, where each process in conflict connects only two (the lowest) interactions in the bipartite graph and only one interaction is selected in every round of coordination. This gives us the upper bound of the number of coordinations for an interaction to be selected: [w/2], where w is the number of interactions connected through processes in conflict in the bipartite graph. This leads to the following theorem about the message complexity of our algorithm.

Theorem 4. Given an *l*-party interaction I, the maximum number of inter-process messages required for I to be selected for execution is 4*l*[w/2], where w is the maximal number of interactions connected through processes in conflict in the bipartite graph of the problem.

5. Summary

First-order multiparty interaction is one of the abstractions in the distributed programming model, called Interacting Processes, proposed by N. Francez and I. R. Forman [3]. In this paper, we presented an algorithm for coordinating first-order multiparty interactions on demand with the middleware support. In this algorithm, middleware serves as the underlying communication infrastructure. Data exchanges are done by middleware. Application developers can develop distributed applications without concerning about the issues of heterogeneity such as data marshalling/unmarshalling and data formats. Applications in different programming languages running on different machines can exchange information across different network systems. In addition, no specific language processor needs to be implemented in order to execute the applications using IP. Our approach allows the applications in any target language to be executed in any general programming environment.

References

[1] C.-C. Chiang and P. Tang, 'Middleware Support for Coordination in Distributed Applications,' *Proceedings of the Fifth IEEE International Symposium on Multimedia Software Engineering (MSE 2003)*, December 10-12, 2003, Taichung: Taiwan, ROC, pp. 148-155.

[2] R. Corchuelo and D. Ruiz, M. Toro, and A. Ruiz, 'Implementing Multiparty Interactions on a Network Computer,' *Proceedings of the XXVth Euromicro Conference*, Milan, Italy, September 1999, pp. 458-465.

[3] N. Francez and I. R. Forman, *Interacting Processes*, Addison-Wesley, 1996.

[4] Y.-J. Joung and S. Smolka, 'A Comprehensive Study of the Complexity of Multiparty Interaction,' *Journal of the ACM*, Vol. 43, No. 1, January 1996, pp. 75-115.