

Interaction and Coordination for Distributed Grid Computing

Peiyi Tang

Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR 72204

James Sinnamon

Department of Computer Science
Australian National University
Canberra ACT 0200 Australia

Abstract

The programming model of the current distributed systems such as middleware CORBA or JAVA RMI is based on remote procedure call. Without the support of high-level API for coordination, developing real-world distributed grid applications in this model is difficult.

In this paper, we present an algorithm for multiparty interaction, a key abstraction of distributed coordination for the future distributed grid programming.

1. Introduction

Grid computing is a new concept and infrastructure in computing inspired by the electric grid which provides pervasive access to power [1]. Like the electric grid, the computing grid provides pervasive access to computing services from heterogeneous and dynamically-changing computer systems distributed in different physical locations.

The concept of grid computing, which is made possible by the fast growth of network capability (it doubles every nine months), opens up many distributed applications and services unthinkable before. One of them is the teleimmersion for education, collaborative work and entertainment. In recent years there has been significant progress made in this area. For example, the Distributed Immersive Performance (DIP) project [6] created a real-time multi-site distributed interactive and collaborative environment for musicians to perform in a common virtual space even though they are in different physical locations. Another example is the distributed chat room for distance education [11]. One of the common requirements to develop the programs for this type of distributed grid computing is the interaction and coordination among the users in the different sites.

The current distributed computing and programming use primarily the client-server model. The application programming interface (API) is mainly based on remote procedure call (RPC). Examples include the middleware CORBA and

the remote method invocation (RMI) in JAVA. The client-server model and RPC programming interface may be sufficient for simple distributed applications such as email, web browsing, online banking, auction and e-commerce, but are not adequate to develop the coordinated grid applications.

We proposed elsewhere to use a high-level programming model of interacting process (IP) based on multiparty interaction to write parallel applications [9]. In this paper, we first argue that the multiparty interaction is a key abstraction to develop future real-world distributed grid applications. We then present a suite of protocols to realize multiparty interaction on top of the message passing machine model. We have implemented our protocols for multiparty interaction in a JAVA package called MULTIPI. MULTIPI, as the core implementation of first-order multiparty interaction, can be used to support any form of distributed grid programming APIs which use multiparty interaction for coordination, synchronization and communication.

The organization of this paper is as follows. In Section 2, we introduce multiparty interaction and show that it is essential to develop the distributed grid applications that require coordination. In Section 3, we present our algorithm of the protocols to realize multiparty interaction. The correctness and message complexity of our protocols are given in Section 4. The design and implementation of MULTIPI are described briefly in Section 5. The paper concludes with the related and future work in Sections 6 and 7, respectively.

2. Writing Distributed Grid Programs with Multiparty Interaction

Interacting Process (IP) is a programming model to develop parallel and distributed applications using multiparty interaction [2].

A program in the IP model is organized in modules called *teams*. A team may contain processes as well as *roles* – formal processes to be enrolled by actual processes from other teams. A multiparty interaction is an object for Ada rendezvous type of synchronization with multiple parties. In the IP model, a process or role can participate in a mul-

tiparty interaction by using an *interaction statement* of format: $a[\vec{v} := \vec{e}]$, where a is the name of the interaction and $\vec{v} := \vec{e}$ the *body* of the interaction for this party. A body is a sequence of assignment statements whose left-hand sides are only local variables of the participating process (role). The interaction statement can be executed only when the controls of all the participating processes have reached the interaction.

Each process or role is a sequences of statements: assignments, `if`, `for` and function or procedure calls as well as interaction statements, and enhanced CSP-like guard selection and iteration statements. The CSP-like guard selection statement in IP is of format:

$$[B_1 \& a_1[\dots] \rightarrow S_1 \square \dots \square B_n \& a_n[\dots] \rightarrow S_n]$$

In the k -th guard ($1 \leq k \leq n$) above, B_k is a boolean predicate of local states called *guarding predicate* and $a_k[\dots]$ an interaction statement called *guarding interaction*; both are optional. A guarding interaction is enabled if the corresponding guarding predicate is true. An enabled guarding interaction can be selected to execute only if it is also enabled by all the other participating processes. Whether it is actually selected is decided through coordination among all the processes involved, because each process may have choices to participate in other interactions. If many guarding interactions in a guard selection statement are enabled, only one of them can be selected to execute. If none of the enabled interactions is selected (due to missing participants for other parties, for example), the process blocks. After a guarding interaction is selected and executed, the sequence of statements denoted by S_k following the right arrow are executed.

The guard iteration statement is similar and of format:

$$*[B_1 \& a_1[\dots] \rightarrow S_1 \square \dots \square B_n \& a_n[\dots] \rightarrow S_n]$$

The difference is that the enclosed guard selection will be executed repeatedly until none of the guarding predicate is true.

To illustrate the power of IP and multiparty interaction, let us write an IP program for a virtual meeting among n philosophers distributed at different hosts in the computing grid. As in a face-to-face meeting in a real conference room, each philosopher repeatedly follows the steps below:

1. Think: Based on the what have been heard, think about what to say next (you may decide not to say anything)
2. Speak or Listen: *Try* to speak if there is something to say and coordinate with all others to decide who to speak. Speak if elected to speak or listen to others otherwise.

Only one philosopher is allowed to speak at a time. When more than one philosophers try to speak, a consensus as who

to speak will be reached through coordination. In an face-to-face meeting, such coordination is done through human interaction. There is no central arbitrator (chair of the meeting) to decide who to speak. Let us call this problem the *speaking philosopher problem*.

The IP program for the speaking philosopher problem using multiparty interaction can be as shown in Figure 1. There are n roles (formal processes) P_i

```

1 team SpeakingPhilosopher(value  $n$ : integer)::
2 [
3    $\parallel_{i=0, n-1}$  role  $P_i$  ::
4     speechi, heardi: message;
5     for (::) {
6       speechi := thinki(heardi);
7       [ (speechi ≠ null) & speaki[] → skip;
8         □ speak0[heardi := speech0] → skip;
9         □ ...
10        □ speaki-1[heardi := speechi-1] → skip;
11        □ speaki+1[heardi := speechi+1] → skip;
12        □ ...
13        □ speakn-1[heardi := speechn-1] → skip;
14      ];
15    }
16]
```

Figure 1. IP Program for Speaking Philosopher Problem

($i = 0, \dots, n - 1$), one for each philosopher, in the team *SpeakingPhilosopher*. The code shown is the program to be executed by process P_i . The codes for other processes P_j ($j \neq i$) are similar and symmetric. Each process P_i ($0 \leq i \leq n - 1$), has two local variables of type **message**, *heard_i* and *speech_i*, for the message heard and the message to speak, respectively. There are n multiparty interactions named *speak_i* ($i = 0, \dots, n - 1$), each of which has n participants: one is the speaker and the others the listeners. In particular, multiparty interaction *speak_i* represents the consensus and the synchronization where process P_i speaks and all the other processes P_j ($j \neq i$) listen to the speech delivered by P_i . *think_i*() at line 6 is the function for P_i to think about what to say next based on what was heard. The coordination for speaking and listening is represented by the guard selection statement in lines 7-14. Since only one enabled multiparty interaction can be selected, it is guaranteed that only one philosopher can speak at a time. The selection of the enabled multiparty interactions is non-deterministic. The bodies (the codes in the brackets) of the guarding inter-

actions are used to carry out data communication. If interaction $speaki$ is selected (line 7), the message in $speech_i$ is broadcast to all the other processes. Likewise, if any of the interactions in lines 8-13 is selected, P_i is to receive the corresponding speech and put it into $heard_i$.

The multiparty interaction in the guard selection or iteration statements en-capsules one of the most common forms of distributed coordination required by real-world distributed grid applications. It is not difficult to see that using remote procedure call and message passing to code such coordination is not only wasteful and unproductive, but also error-prone.

3. Coordination Protocols for Multiparty Interaction

In this section, we present our algorithm for coordination protocols to realize multiparty interaction using message passing communication.

3.1. Problem Definition

We assume that there are m interacting formal processes P_j ($j = 1, \dots, m$) trying to coordinate through n multiparty interactions I_i ($i = 1, \dots, n$). The goal of coordination is to select (non-deterministically) a subset of the multiparty interactions for execution, subject to the following constraints:

1. Each interaction selected for execution must have all of its parties participated by distinct processes.
2. No process can participate in executions of more than one 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. This problem is also known as guard scheduling problem.

3.2. Protocols

Each multiparty interaction I_i needs a proxy process, also denoted as I_i , to coordinate the participating processes. When a process is ready to participate in its enabled multiparty interactions, it starts a separate thread for each of them. Let there be k_j enabled interactions, $I_{i_1}, \dots, I_{i_{k_j}}$, in which process P_j is ready to participate. The thread started by P_j to interact with interaction I_{i_r} ($1 \leq r \leq k_j$) is denoted as $T_{i_r}^j$. In addition, process P_j also starts a master thread denoted as M^j to manage the coordination among threads $T_{i_r}^j$ and the communication between threads $T_{i_r}^j$ and threads I_{i_r} .

```

*[ 1.0
  state = 'init' → send Request( $P_j, p$ ) to  $I_{i_r}$ ;
  state := 'req-sent'
□ 1.1
  state = 'req-sent'; receive All-Met from  $I_{i_r}$  →
  synchronized( $a[]$ ) {
    if ( $a[1..k] = 0$ )
       $a[r] := 1$ ; send Commit( $P_j, p$ ) to  $I_{i_r}$ ;
      state := 'commit-sent'
    else if ( $a[1..(r-1)] \neq 0 \wedge a[(r+1)..k] = 0$ )
      send Withdraw( $P_j, p$ ) to  $I_{i_r}$ ;
      send Re-try( $T_{i_r}^j$ ) to  $M^j$ ;
      state := 're-try'
    else if ( $a[(r+1)..k] \neq 0$ )
       $a[r] := 1$ ; state := 'pending'
    endif
  }
□ 1.2
  state = 'req-sent'; receive Stop from  $M^j$  →
  send Abort( $P_j, p$ ) to  $I_{i_r}$ ;
  send Ready-to-Die( $T_{i_r}^j$ ) to  $M^j$ ;
  state := 'ready-to-die'
□ 1.3
  state = 'pending'; receive Continue from  $T_{i_{r'}}^j$  ( $r < r'$ ) →
  send Commit( $P_j, p$ ) to  $I_{i_r}$ ; state := 'commit-sent'
□ 1.4
  state = 'pending'; receive Stop from  $M^j$  →
  send Withdraw( $P_j, p$ ) to  $I_{i_r}$ ;
  send Ready-to-Die( $T_{i_r}^j$ ) to  $M^j$ ;
  synchronized( $a[]$ ) {  $a[r] := 0$  }; state := 'ready-to-die'
□ 1.5
  state = 'commit-sent';
  receive Succeed( $P_{j_1}, \dots, P_{j_q}$ ) from  $I_{i_r}$  →
  record process list ( $P_{j_1}, \dots, P_{j_q}$ );
  send Finish( $T_{i_r}^j$ ) to  $M^j$ ;
  state := 'success'
□ 1.6
  state = 'commit-sent'; receive Fail from  $I_{i_r}$  →
  synchronized( $a[]$ ) {
     $a[r] := 0$ ;
    if ( $a[1..(r-1)] \neq 0$ )
      let  $r'$  be the largest integer such that
         $1 \leq r' \leq (r-1) \wedge a[r'] = 1$ ;
      send Continue to  $T_{i_{r'}}^j$ ;
    endif
    send Re-Try( $T_{i_r}^j$ ) to  $M^j$ ; state := 're-try'
  }
□ 1.7
  state = 're-try'; receive Stop from  $M^j$  →
  send Ready-to-Die( $T_{i_r}^j$ ) to  $M^j$ ; state := 'ready-to-die'
□ 1.8
  state = 're-try'; receive Try-Again from  $M^j$  →
  state := 'init'
□ 1.9
  state = 'ready-to-die'; receive Kill from  $M^j$  → kill itself
]

```

Figure 2. Protocol of Thread $T_{i_r}^j$

We assume that the underlying communications between all the processes and threads are asynchronous and carried out via reliable FIFO channels. Each thread or process also maintains a queue for incoming messages.

Our guard scheduling algorithm consists of three protocols for $T_{i_r}^j$, I_{i_r} and M^j . Thread $T_{i_r}^j$ communicates with process I_{i_r} and thread M^j . Threads $T_{i_r}^j$ ($r = 1, \dots, k_j$) also communicate with each other. The pseudo codes of the protocols for $T_{i_r}^j$, I_{i_r} and M^j are shown in Figures 2, 3 and 4, respectively.

In this paper, we use CSP-like iterative guard commands as follows to describe the communication protocols:

$$*[g_1 \rightarrow S_1 \square \dots \square g_n \rightarrow S_n]$$

Guard commands are referred to as *rules* in this paper. All the rules in the three protocols are numbered for the ease of reference.

The basic idea behind the algorithm is the two-phase locking. $T_{i_r}^j$ first sends a request with its identity to I_{i_r} and then waits for a message called *All-Met* from it. I_{i_r} records the request from $T_{i_r}^j$ and will not send *All-Met* back until it receives all the l_{i_r} requests it needs. The protocol of $T_{i_r}^j$ enters the second phase upon receiving *All-Met*.

Depending on the states of other T^j threads of P_j , $T_{i_r}^j$ can either send a commitment or withdrawal to I_{i_r} , or suspend the decision until late. Only one thread $T_{i_r}^j$ of P_j is allowed to send commitment at a time. After receiving commitment from $T_{i_r}^j$, I_{i_r} either sends message *Fail* back to $T_{i_r}^j$ if it has ever received a withdrawal from its participating processes, or sends message *Success* if it has received the commitments from all of them. Upon receiving message *Success*, $T_{i_r}^j$ signals M^j to kill all other T^j threads of P_j . If $T_{i_r}^j$ receives message *Fail*, it continues to go to the next round of coordination.

We assume that there is a total order among all the interactions involved. Without loss of generality, we assume $I_1 < \dots < I_n$ and give the highest priority to I_1 . To simplify the notation, we use k to denote k_j . Among the k interactions, I_{i_1}, \dots, I_{i_k} , we assume $I_{i_1} < \dots < I_{i_k}$, i.e. $i_1 < \dots < i_k$. This total order is essential both to prevent deadlock and to ensure progress in each round of coordination.

Process P_j maintains a bit map $a[1..k]$ ¹ shared by all its threads $T_{i_r}^j$ ($r = 1, \dots, k$). The initial value of every bit of $a[1..k]$ is 0. Bit $a[r]$ is set to 1 when thread $T_{i_r}^j$ has either committed P_j to the corresponding interactions I_{i_r} , or is in state '*pending*'. A pending thread will eventually com-

```

* [ 2.0
  state = 'meeting'; receive Request( $P_j, p$ ) from  $T_{i_r}^j \rightarrow$ 
  record  $P_j$  as the participant for party  $p$ ;  $nReq + +$ ;
  if ( $nReq = q$ )
     $nReq := 0$ ;
    for each of the  $q$  recorded  $P_j$  send All-Met to  $T_{i_r}^j$ ;
    state := 'all-met'
  endif
□ 2.1
state = 'meeting'; receive Abort( $P_j, p$ ) from  $T_{i_r}^j \rightarrow$ 
discard  $P_j$  as the participant for party  $p$ ;
 $nReq - -$ ;
□ 2.2
state = 'all-met'; receive Commit( $P_j, p$ ) from  $T_{i_r}^j \rightarrow$ 
if ( $nW = 0$ )
  record  $P_j$ 's commitment for party  $p$ ;
   $nC + +$ ;
  if ( $nC = q$ )
    let  $P_{j_1}, \dots, P_{j_q}$  be the  $q$  recorded
       $P_j$  with commitment;
    for ( $k = 1$  to  $q$ ) send Succeed( $P_{j_1}, \dots, P_{j_q}$ ) to  $T_{i_r}^{j_k}$ ;
     $nC := 0$ ;
    state := 'success'
  endif
else
  send Fail to  $T_{i_r}^j$ ;
   $nC + +$ ;
  if ( $nC + nW = q$ )
     $nW := 0$ ;  $nC := 0$ ;
    discard all the records;
    state := 'meeting'
  endif
endif
□ 2.3
state = 'all-met'; receive Withdraw( $P_j, p$ ) or
Abort( $P_j, p$ ) from  $T_{i_r}^j \rightarrow$ 
if ( $nW = 0 \wedge nC \neq 0$ )
  let  $P_{j_1}, \dots, P_{j_{nC}}$  be the  $nC$  recorded
     $P_j$  with commitment;
  for ( $k = 1$  to  $nC$ ) send Fail to  $T_{i_r}^{j_k}$ ;
endif;
 $nW + +$ ;
if ( $nC + nW = q$ )
   $nW := 0$ ;  $nC := 0$ ;
  discard all the records;
  state := 'meeting'
endif
]

```

Figure 3. Protocol of Interaction Process I_{i_r}

1 In this paper, $a[1..k]$ denotes an array a of k elements with the index ranging from 1 to k . $a[i..j]$ ($1 \leq i < j \leq k$) is a subarray of $a[1..k]$ from the i -th element to the j -th element. $a[r]$ ($1 \leq r \leq k$) denotes the r -th element of $a[1..k]$.

mit to its corresponding interaction unless it is stopped and killed.

Since the bit map $a[1..k]$ is shared by all the threads of process P_j , its accesses should be put into critical sections. We use *synchronized* block as in JAVA to indicate critical sections.

Message *Continue* is sent by $T_{i_r}^j$ to wake up a pending thread, upon receiving *Fail* from I_{i_r} (rule 1.6). Upon receiving *Continue*, the pending thread proceeds to send *Commit*() to its own interaction (rule 1.3).

In the protocol of I_{i_r} , we use q to denote l_{i_r} , the number of parties of interaction I_{i_r} .

Assuming that there is no deadlock in the protocol of $T_{i_r}^j$ (It will be proved in Section 4.2.), $T_{i_r}^j$ will send either a *Commit*() or a *Withdraw*() to I_{i_r} , after receiving *All-Met* from it. I_{i_r} may receive *Abort*() from $T_{i_r}^j$ in state '*all-met*'. This is because $T_{i_r}^j$ may receive *Stop* from its M^j in state '*req-sent*' (due to the fact that another thread succeeds) before the message *All-Met* from I_{i_r} reaches it. This is the reason why I_{i_r} should be prepared to accept *Abort*() in state '*all-met*' (rule 2.3). An *Abort*() received in state '*all-met*' is treated as a *Withdraw*(). After I_{i_r} receives the responses from all the participating processes, it enters state '*success*' if it receives all *Commit*()s, or goes back to state '*meeting*' for the next round of coordination otherwise.

The main function of protocol of thread M^j in Figure 4 is to coordinate all the threads $T_{i_1}^j, \dots, T_{i_k}^j$. It also intercepts and relays messages between $T_{i_r}^j$ and I_{i_r} . In particular, it discards all the messages to $T_{i_r}^j$ if it has killed $T_{i_r}^j$.

Thread M^j maintains a counter, $nRetry$, to synchronize all the threads before entering the next round of coordination. The next round of coordination should not start until all the threads $T_{i_r}^j$ ($r = 1, \dots, k$) fail (rule 3.1).

After one of threads $T_{i_r}^j$ succeeds, thread M^j is responsible to send *Stop* to all the other threads (rule 3.2). Another counter, $nDied$, is used to make sure that all the other threads are killed before M^j enters state '*success*' (rule 3.3).

4. Correctness and Complexity

In this section, we prove the correctness and analyse 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*.

```

*[ 3.0
  state = 'init' →
  spawn k threads  $T_{i_1}^j, \dots, T_{i_k}^j$  in 'init' state;
  state := 'working'; nRetry := 0; nDied := 0
□ 3.1
  state = 'working'; receive Re-Try( $T_{i_r}^j$ ) →
  nRetry ++;
  if (nRetry = k)
    nRetry := 0;
    send Try-Again to all  $T_{i_1}^j, \dots, T_{i_k}^j$ 
  endif
□ 3.2
  state = 'working'; receive Finish( $T_{i_r}^j$ ) →
  send Stop to all  $T_{i_1}^j, \dots, T_{i_{r-1}}^j, T_{i_{r+1}}^j, \dots, T_{i_k}^j$ ;
  state := 'finishing'
□ 3.3
  state = 'finishing'; receive Ready-to-Die( $T_{i_r}^j$ ) →
  send Kill to  $T_{i_r}^j$ ;
  nDied ++;
  if (nDied = k - 1)
    state := 'success'
  endif
]

```

Figure 4. Protocol of Thread M^j

4.1. Safety

The safety requirement of the guard scheduling problem defined in Section 3.1 demands that

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

The interaction safety requirement can be derived from the protocol of I_{i_r} directly. In particular, process I_{i_r} will not enter state '*all-met*' unless it receives the requests for participation from q (i.e. l_{i_r}) processes. Furthermore, it will not enter 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 threads, $T_{i_1}^j, \dots, T_{i_k}^j$, started by P_j , only one can enter state '*success*'.

Proof: Thread $T_{i_r}^j$ can enter state '*success*' only from state '*commit-sent*'. It can enter state '*commit-sent*' either from state '*pending*' or state '*request-sent*'. Thread $T_{i_r}^j$ moves from state '*request-sent*' to state '*commit-sent*' only when all the bits of bit map $a[1..k]$ are 0 (rule 1.1). If it moves into state '*pending*', it will not enter state '*commit-sent*' until another thread sends it a *Continue* after leaving state '*commit-sent*' (rule 1.6). Therefore, there

is only one thread in state '*commit-sent*' at any time. After the thread enters state '*success*', all other threads will be killed (rules 1.5 \Rightarrow 1.2, 1.4, 1.7 \Rightarrow 3.3 \Rightarrow 1.9). \square

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 $T_{i_r}^j$, M^j and I_{i_r} . In particular, no process or thread is allowed to stay in a waiting state indefinitely.

After I_i receives all the l_i requests it needs and enters state '*all-met*', it will receive the same number (l_i) of *Commit()*, *Withdraw()* or *Abort()* (rules 1.1, 1.2, 1.3 and 1.4), provided that each thread $T_{i_r}^j$ involved is live and responds eventually. After that, I_i will either enter state '*meeting*' again for the next round of coordination or enter state '*success*'. In other words, I_i is live as long as each thread $T_{i_r}^j$ with which it communicates is live.

Similarly, if every thread $T_{i_r}^j$ ($1 \leq r \leq k$) 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 threads $T_{i_r}^j$ ($r = 1, \dots, k$) are unsuccessful (rules 1.1 and 1.6). If one thread succeeds, M^j will receive *Finish()* from it (rule 1.5) and enter state '*finishing*'. M^j will further receive $(k - 1)$ *Ready-to-Die()*s from the remaining threads (rules 1.2, 1.4 and 1.7) and enter state '*success*'.

Therefore, the liveness of the entire system hinges on the liveness of the protocol of $T_{i_r}^j$. The following lemma is used to prove the liveness of $T_{i_r}^j$.

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

Proof: According to rule 1.1, $a[r] = 1$ only if $T_{i_r}^j$ is in states '*commit-sent*' or '*pending*', but the first thread $T_{i_r}^j$ with $a[r] = 1$ must be in state '*commit-sent*'.

To simplify the notation, we rename $T_{i_r}^j$ to be $T_{i_r}^j$. Let us assume that $T_{i_r}^j$ stays in state '*pending*' indefinitely.

When thread $T_{i_r}^j$ enters state '*pending*', $a[(r + 1)..k] \neq 0$ must be held. Let $a[u_1], \dots, a[u_v]$ ($r + 1 \leq u_1 < \dots < u_v \leq k$) be all the bits that either are 1 when $T_{i_r}^j$ enters state '*pending*' or ever become 1 while $T_{i_r}^j$ is in state '*pending*' (indefinitely).

Thread $T_{i_{u_v}}^j$ must be in state '*commit-sent*' when $T_{i_r}^j$ enters state '*pending*'. Other threads $T_{i_{u_1}}^j, \dots, T_{i_{u_{v-1}}}^j$ must be in state '*pending*' first.

We want to prove that based on the assumption above at least one of $T_{i_{u_1}}^j, \dots, T_{i_{u_v}}^j$ must be in state '*commit-sent*' indefinitely.

Consider thread $T_{i_{u_v}}^j$ 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()*, $T_{i_r}^j$ would leave state '*pending*' in finite time (rules 1.5 \Rightarrow 3.2 \Rightarrow 1.4). This would contradict the assumption above.

If it receives a *Fail*, thread $T_{i_{u_{v-1}}}^j$ will enter state '*commit-sent*' in finite time (rules 1.6 \Rightarrow 1.3).

The same procedure will also apply to threads $T_{i_{u_{v-1}}}^j, \dots, T_{i_{u_1}}^j$. Therefore, if none of $T_{i_{u_1}}^j, \dots, T_{i_{u_v}}^j$ can stay in state '*commit-sent*' indefinitely, $T_{i_r}^j$ will leave state '*pending*' in finite time. This proves the lemma. \square

There are four waiting states in the protocol of $T_{i_r}^j$: '*req-sent*', '*commit-sent*', '*pending*' and '*re-try*'. The waiting of $T_{i_r}^j$ in state '*req-sent*' is to ensure interaction safety and should not be considered as a problem for liveness.

$T_{i_r}^j$ in state '*re-try*' will enter state '*init*' after all the threads started by P_j send *Withdraw()*s to their interactions (rules 1.1, 1.4 \Rightarrow 3.1 \Rightarrow 1.8). Therefore, for the liveness of the protocol of $T_{i_r}^j$, we only need to prove that no thread $T_{i_r}^j$ will stay in states '*commit-sent*' or '*pending*' indefinitely. This is done in the following theorem.

Theorem 2 *It is impossible for any thread $T_{i_r}^j$ in the system to stay in states '*commit-sent*' or '*pending*' indefinitely.*

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

Let us assume that there is a thread $T_{i_1}^{j_1}$ staying in state '*commit-sent*' indefinitely. This means that $T_{i_1}^{j_1}$ receives neither *Success()* nor *Fail* in finite time. Therefore, none of the threads coordinating interaction I_{i_1} ever sends a *Withdraw()* or an *Abort()* to it (rules 2.2, 2.3). Furthermore, there is at least one of these threads that does not ever send a *Commit()* either. Let this thread be $T_{i_1}^{j_2}$. According to the protocol, $T_{i_1}^{j_2}$ must be in state '*pending*' indefinitely. From Lemma 1, there must be another thread $T_{i_2}^{j_2}$ from the same process P_{j_2} such that it stays in state '*commit-sent*' indefinitely and $i_1 < i_2$.

Continuing this way, we will have an infinite series

$$T_{i_1}^{j_1}, T_{i_1}^{j_2}, T_{i_2}^{j_2}, \dots, T_{i_{k-1}}^{j_k}, T_{i_k}^{j_k}, \dots$$

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

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 state *'meeting'* or state *'success'* in finite time. The progress requirement demands that at least one of those interactions in state *'all-met'* enter state *'success'*.

This requirement is satisfied in our algorithm. In order to prove this, we need the lemma as follows.

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

Proof: We prove this lemma by construction. Thread $T_{i_r}^j$ in state *'req-sent'* sends a $Withdraw()$ to I_{i_r} only if it finds $a[1..(r-1)] \neq \mathbf{0}$ (rule 1.1). Let r' be the largest integer such that $r' < r$ and $a[r'] = 1$. According to the protocol, thread $T_{i_{r'}}^j$ is either the first thread in state *'commit-sent'* or a past pending thread which has been woken up by another thread and entered state *'commit-sent'* (rule 1.3). \square

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_{u_1}, \dots, I_{u_w} be the subset of all the interactions that enter state *'all-met'* after receiving all the requests they need. Then, at least one of them will enter state *'success'*.*

Proof: Let P_{v_1}, \dots, P_{v_y} be all the processes involved to make I_{u_1}, \dots, I_{u_w} enter state *'all-met'*. Without loss of generality, we also assume $I_{u_1} < \dots < I_{u_w}$, i.e. $u_1 < \dots < u_w$.

Due to the liveness of the system, every interaction of I_{u_1}, \dots, I_{u_w} will receive a response, $Commit()$, $Withdraw()$, or $Abort()$, from each of its participating processes from P_{v_1}, \dots, P_{v_y} and enter either state *'meeting'* or state *'success'*. Let us assume that none of I_{u_1}, \dots, I_{u_w} enters state *'success'*.

According to the protocol, a thread $T_{u_i}^{v_j}$ ($1 \leq j \leq y, 1 \leq i \leq w$) can send $Withdraw()$ only in two states: *'req-sent'* (rule 1.1) and *'pending'* (rule 1.4). But, based on the assumption above, it is impossible for $T_{u_i}^{v_j}$ to send $Withdraw()$ in state *'pending'*. This is because otherwise it must receive a *Stop* from M^{v_j} and therefore there must be a thread $T_{u_{i'}}^{v_j}$ ($1 \leq i' \leq w$) that succeeds in its coordination (rules $1.5 \Rightarrow 3.2 \Rightarrow 1.4$). This would imply that $I_{u_{i'}}$ enters state *'success'*.

To simplify the notation, P^{v_j} , I_{u_i} and $T_{u_i}^{v_j}$ are renamed P^{j_1} , I'_{i_1} , $T^{j_1}_{i_1}$, respectively.

Consider I'_w first. Because it enters state *'meeting'*, it must have received at least one $Withdraw()$ from, say, $T'^{j_1}_{i_1}$ ($1 \leq j_1 \leq y$). According to Lemma 2, there must be a thread $T'^{j_1}_{i_1}$ that has sent a $Commit()$ to I'_{i_1} such that $i_1 < w$. Since I'_{i_1} also enters state *'meeting'*, it must have received a $Withdraw()$ from, say, $T'^{j_2}_{i_2}$ ($1 \leq j_2 \leq y$). By using Lemma 2 again, we can find another thread $T'^{j_2}_{i_2}$ that has sent a $Commit()$ to I'_{i_2} such that $i_2 < i_1$. Continuing this way, we will have an infinite series

$$T'^{j_1}_w, T'^{j_1}_{i_1}, T'^{j_2}_{i_1}, \dots, T'^{j_k}_{i_k}, T'^{j_{k+1}}_{i_k}, \dots$$

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

4.4. Message Complexity

In our algorithm, I_{i_r} 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.

The message complexity our algorithm is given by the following theorem.

Theorem 4 *Given an l -party interaction I , the maximum number of interprocess messages required for I to be selected for execution is $4l \lfloor \frac{w}{2} \rfloor$, where w is the maximal number of interactions connected through processes in conflict.*

Here processes in conflict are the processes trying to participate more than one interactions. This complexity is less than the algorithm proposed in [3]. The details of the analysis can be found in [8].

5. Implementation of Multiparty Interaction Coordination

As the first step towards the future programming model and system for distributed grid computing, we implemented the coordination of multiparty interaction in a JAVA package called MULTIPI.

5.1. Problem Cast

Historically, the zero-order multiparty interaction (where each party can be participated by a fixed process) scheduling problem was cast as the *committee coordination* problem.

In our MULTIPI, the coordination of first-order multiparty interaction is cast to a *movie production coordination* problem.

5.2. Design and Implementation of MULTIPI

The purpose of MULTIPI is to implement and experiment with the coordination protocols of multiparty interaction described in Section 3.

A movie production problem is specified in a XML file. The `main()` method of MULTIPI uses JDOM and xerces to parse this file to create a tree, which is then passed to the constructor of a single `Coordinator` thread. The `Coordinator` thread is mainly to manage the rounds of coordinations between `Interactor` threads and `InteractionManager` threads after it constructs them.

The major classes in MULTIPI are as follows:

- **Interactor:** This is a thread running as an IP process which executes a guard selection statement. In the problem cast of movie production, it is the process for each individual actor.
- **Bidder:** This is a thread running the protocol of $T_{i_r}^j$ in Section 3. The protocol is implemented in the `updateState()` method of the class.
- **InteractionManager:** This is a thread running the protocol of I_{i_r} in Section 3 for each multiparty interaction. In the problem cast of movie production, it represents a movie production. The protocol of I_{i_r} is implemented in the `coordinate()` method.

MULTIPI is currently implemented in a JAVA package of 3505 lines running on a single JVM. We are currently extending it to run on multiple JVMs using dJVM [10]. The distribution of the current MULTIPI version 0.3 can be obtained from [7].

6. Related Work

The first coordination algorithm for first-order multiparty interaction was proposed by Joung and Smolka [3]. The message-complexity of our algorithm is significantly less[8].

There were very few attempts to implement first-order multiparty interaction to be used for distributed programming. The closest ones are in [5, 12, 4]. But none of them provides the algorithm or the source code of the implementation.

7. Future Work

We will experiment with MULTIPI after we extend it to multiple JVMs using dJVM [10]. We will address the fair-

ness issue by randomizing the total order of the interactions. Finally, we will design and implement a distributed grid programming language incorporating multiparty interaction in MULTIPI as the basic constructor for distributed coordination.

References

- [1] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999.
- [2] N. Francez and I. R. Forman. *Interacting Processes – A Multiparty Approach to Coordinated Distributed Programming*. Addison-Wesley, 1996.
- [3] Y.-J. Joung and S. A. Smolka. Coordinating first-order multiparty interactions. *ACM Transactions on Programming Languages and Systems*, 16(3):954–985, May 1994.
- [4] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. Coordinated atomic actions: From concept to implementation. Technical Report 595, 1997.
- [5] A. Ruiz, R. Corchuelo, J. Duan, and M. Toro. An aspect-oriented approach based on multiparty interactions to specifying the behavior of a system. In *Proceedings of the International Conference on Principles, Logics, and Implementation of High-Level Programming Languages (PLI99)*, pages 56–65, Paris, France, Oct. 1999.
- [6] A. A. Sawchuk, E. Chew, R. Zimmermann, C. Papadopoulos, and C. Kyriakakis. From remote media immersion to distributed immersive performance. In *Proceedings of the ACM SIGMM 2003 Workshop on Experiential Telepresence*, Nov. 2003.
- [7] J. Sinnamon. <http://titus.compsci.ualr.edu/~ptang/multipi/>.
- [8] P. Tang and Y. Muraoka. On-demand coordination of first-order multiparty interactions. Faculty of Sciences Working Paper Series (<http://www.sci.usq.edu.au/research/>) SC-MC-9902, Department of Mathematics and Computing, University of Southern Queensland, Jan. 1999.
- [9] P. Tang and Y. Muraoka. Parallel programming with interacting processes. In *Proceedings of the 12-th International Workshop on Languages and Compilers for Parallel Computing (LCPC99)*, Lecture Notes in Computer Science, 1863, pages 201–218, San Diego, USA, Aug. 1999.
- [10] J. Zigman and R. Sankaranarayana. djvm - a distributed jvm on a cluster. Technical Report TR-CS-02-04, Department of Computer Science, Australian National University, 2002.
- [11] R. Zimmermann, B. Seo, L. S. Liu, R. S. Hampole, and B. Nash. Audiopeer: A collaborative distributed audio chat system. In *Proceedings of the Tenth International Conference on Distributed Multimedia Systems (DSM'04)*, Sept. 2004.
- [12] A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. *ACM SIGPLAN Notice*, 34(10):435–446, 1999.