

On the Design of Heterogeneous Applications for Distributed Coordination

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

Abstract

The current state of the art in the existing middleware technologies does not support the development of distributed applications that need processes to complete a task collaboratively. What is needed in the next generation of middleware is a synergy of heterogeneity, distribution, communication, and coordination. We are proposing to augment the existing middleware technologies to provide collaboration support through a multiparty interaction (MI) protocol rather than design a new programming language for distributed coordinated programming. In this paper, a 4-layered interaction model will be presented to decouple the applications and their underlying middleware implementations including coordination protocols by providing a set of generic interfaces to the applications. The decoupling of applications and middleware technologies by isolating computation, communication, and coordination promotes reuse, improves comprehension, and eases maintenance due to software evolution.

1. Introduction

The existing middleware technologies have been used to aid the development of distributed applications in heterogeneous computing environments. However, the current state of the art in the existing middleware technologies mainly supports client-server programming model. The language construct supported is, by large, remote procedure call (RPC). This simple RPC-based client-server programming model is not adequate to develop distributed applications that need three or more processes to work together collaboratively.

To support the development of heterogeneous distributed applications for coordination, we are proposing to augment the existing middleware technologies to provide collaboration support through a multiparty interaction protocol rather than design a new programming language for distributed coordinated programming. A 4-layered interaction model will be presented to decouple the applications and their underlying middleware

implementations including coordination protocols by providing a set of generic interfaces to the applications.

2. A 4-layered model for heterogeneous distributed coordination

Radestock and Eisenbach [5] present the concerns in developing coordinated distributed applications that interact with each other. The concerns are separated into four parts: the communication part, the computation part, the configuration part, and the coordination part. The communication part defines how components communicate with each other. The computation part defines the behavior of a component which also determines what is being communicated. The configuration part indicates which components exist, which components can communicate with each other, the method of communication, where data come from, and where the data are sent to. The coordination part determines when certain interactions will occur. The dependencies of these four concerns yield a 4-layered protocol structure in Figure 1.

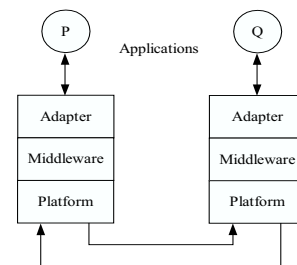


Figure 1. Structure of communicating components through adapters

The model shown in Figure 1 facilitates separation of concerns by allowing for a layered development, in which a layer achieving one concern is superimposed over another layer achieving another concern. From a software engineering point of view, lower layers need not, and should not, know about the higher layers. Each layer will have a collection of well-defined functions that access the

layer below it by using its interface. This clear separation of concerns is extremely beneficial, enabling a high degree of reuse, and easier maintenance. In the following subsections, each layer will be discussed in detail.

2.1 Development of distributed applications for coordination

IP [3] is the programming language adopted to develop heterogeneous distributed coordinated applications in our model. We argue that developers should only develop IP programs for coordinated distributed applications, and nothing more. In this research, we are not implementing a new language processor to execute IP programs. On the contrary, our intention is to allow IP programs to be realized under any general programming environment. The approach we use is to analyze an IP program and generate a multiparty interaction description that is a data structure describing 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 program. Application developers then write a program in the target language to include the multiparty interaction description in the program. A function in the Adapter layer will be invoked to represent the caller (participating party) to interact with other participants for coordination. A set of APIs will be developed to facilitate the process to obtain the multiparty interaction information in the multiparty interaction description at run-time.

2.2 The adapter layer

Applications built in the existing middleware technologies suffer from the complexity of high interactions in components. A lack of component independence is a problem for developing reusable components. In our model, a component's interactions interfacing with its underlying middleware including coordination will be encapsulated and isolated outside the component so that the component can be built to be independent of the context in which it is used, allowing it to be reused in many different computing environments.

2.2.1 Coordination support in the adapter layer

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.

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 manage 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.

2.2.2 Open communication through middleware

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)*. The implementation of *GetData()* and *PutData()* is placed in the Adapter layer of the protocol structure shown in Figure 1. *GetData()* and *PutData()* transmit data through the invocation of middleware functions implemented in the Middleware layer. At this moment, no participating processes can continue until they all have completed data exchanges.

The format of the *OperationArgumentBuffer* is defined in the interface description in the target language. The interface description is compiled from an IP that describes the interfaces of a coordination module. Every participating process includes a copy of the interface description in the code. Before any interaction occurs, the participating processes must register their interfaces to their associated adapters so the adapters can resolve the signature of operations at runtime, such as *GetData()* and *PutData()*. This dynamic interface binding was designed to allow an adapter to examine the signature of the requested services at runtime such as operation names, parameters orders, parameters types, and parameters sizes. In addition, the interface language mapping allows an interface in a specific programming language to be automatically generated from an IP program.

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 ($i=1, \dots, n$), each of which has l_i parties to be participated by distinct processes form m participating processes P_j ($j=1, \dots, m$) whose identifiers are not known until run-time, the guard scheduling problem is to select a 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 one interaction.
3. If there are interactions that 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 is the liveness requirement of the problem.

For each interaction I_i , we create an interaction process, denoted as I_i . If P_j is ready to participate in k interactions I_{i1}, \dots, I_{ik} , we create (1) a thread manager M_j and (2) one proxy thread $T_{j,ir}$ for each of I_{i1}, \dots, I_{ik} . The proxy thread $T_{j,ir}$ is used to communicate with the interaction process, I_{ir} . A thread manager M_j serves as the manager of all the proxy threads $T_{j,i1}, \dots, T_{j,ik}$ that it spawns.

The basic idea of the protocol for $T_{j,ir}$ is as follows: It sends a *Request* message to I_{ir} to notify its intention to participate. When I_{ir} receives all the *Request* messages 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 the 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 a transition to the “commit-sent” state; (2) if a $T_{j,ir'}$ with higher priority has committed to I_{ir} , $T_{j,ir}$ withdraws its participation by sending a *Withdraw* message to I_{ir} and makes a transition to the “re-try” state, or (3) if a $T_{j,ir'}$ with lower priority has committed to I_{ir} , $T_{j,ir}$ makes a transition to the “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 the “commit-sent” state, $T_{j,ir}$ waits for a *Succeed* message from I_{ir} when it receives commitment from all of its participants. After $T_{j,ir}$ receives the *Succeed* message, it sends a *Finish* message to its thread manager M_j to register the activation of I_{ir} and makes a transition to the “success” state.

The protocol for the 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 a *Request* message or an *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 an *All-met* message to all of the participants, and makes a transition to the “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 the “success” state (when all participants committed) or the “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 protocol for the thread manager M_j is to coordinate all the proxy threads. It also intercepts and relays 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 $T_{j,i1}, \dots, T_{j,ik}$. After it spawns the proxy threads $T_{j,i1}, \dots, T_{j,ik}$, it waits for either a *Retry* message or a *Finish* message from each of them. Upon receiving the *Finish* message from $T_{j,ir}$, it sends a *Stop* message to all the other proxy threads so that they can send *Withdrawal* or *Abort* messages to their corresponding interaction managers before making transitions to the “ready-to-die” state. If all proxy threads send *Retry* messages, M_j sends a *TryAgain* message back and allow them to start the next round of coordination. The details of the above three protocols can be found in [1].

4. Correctness

In this section, we prove the correctness 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 demands that

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

The interaction safety requirement can be derived from the protocol of I_r directly. In particular, the process I_r will not enter into the ‘all-met’ state unless it receives the requests for participation from q (i.e. l_r) processes. Furthermore, it will not enter into the ‘success’ state 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 $T_{j,i1}, \dots, T_{j,ik}$, started by P_j , only one can enter into the ‘success’ state.

Proof: A thread $T_{j,ir}$ can enter into the ‘success’ state only from the ‘commit-sent’ state. It can enter into the ‘commit-sent’ state either from the ‘pending’ state or the ‘request-sent’ state. The thread $T_{j,ir}$ moves from the ‘request-sent’ state to the ‘commit-sent’ state only when all the bits of bit map $a[]$ are 0’s. If it moves into the ‘pending’ state, it will not enter into the ‘commit-sent’ state until another thread sends it a *Continue* message after leaving the ‘commit-sent’ state. Therefore, there is only one thread in the ‘commit-sent’ state at any time. After the thread enters into the ‘success’ state, 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_i . In particular, no process or thread is allowed to stay in a waiting state indefinitely.

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

Similarly, if every thread $T_{j,ir}$ ($1 \leq r \leq k$) is live, the thread M_j is also live. In particular, the thread M_j will remain in the ‘working’ state and start the next round of coordination if all the k proxy threads $T_{j,ir}$ ($r = 1, \dots, k$) are successful. If one thread succeeds, M_j will receive *Finish* from it and enter into the ‘finishing’ state. M_j will further receive $(k-1)$ *ReadyToDie* messages from the remaining threads and enters into the ‘success’ state. 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 thread $T_{j,ir}$ is in the ‘pending’ state indefinitely, there must be another thread $T_{j,ir}$ of P_j such that $r < r'$ in the ‘commit-sent’ state indefinitely.

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

When the thread $T_{j,ir}$ enters into the ‘pending’ state, $a[(r+1) \dots k] \neq 0$ 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’s when $T'_{j,r}$ enters into the ‘pending’ state or ever become 1’s when $T'_{j,r}$ is in the ‘pending’ state indefinitely.

The thread $T'_{j,uv}$ must be in the ‘commit-sent’ state when $T'_{j,r}$ enters into the ‘pending’ state. Other threads

$T'_{j,u1}, \dots, T'_{j,uv-1}$ must be in the ‘pending’ state first. We want to prove that based on the assumption above at least one of $T'_{j,u1}, \dots, T'_{j,uv}$ must be in the ‘commit-sent’ state indefinitely.

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

There are four waiting states in the protocol of $T_{j,ir}$: ‘request-sent’, ‘commit-sent’, ‘pending’, and ‘re-try’. The waiting of $T_{j,ir}$ in the ‘request-sent’ state is to ensure interaction safety and should not be considered as a problem for liveness. $T_{j,ir}$ in the ‘re-try’ state will enter into the ‘init’ state after all the threads started by P_j send their *Withdraw* messages to their interaction processes. 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 the ‘commit-sent’ state or the ‘pending’ state indefinitely. This is done in the following theorem.

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

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

Let us assume that there is a thread $T_{j1,i1}$ staying in the ‘commit-sent’ state indefinitely. This means that $T_{j1,i1}$ receives neither *Succeed* nor *Fail* in finite time. Therefore, none of the threads coordinating interaction I_{i1} ever sends a *Withdraw* message or an *Abort* message to it. Furthermore, there is at least one of these threads that does not ever send a *Commit* message 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 the ‘commit-sent’ state indefinitely and $i_1 < i_2$. Continuing this way, we will have an infinite series

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

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 $i_1 < i_2 < \dots < i_k < \dots$. 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.

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 the ‘all-met’ state will enter into the ‘meeting’ state or the ‘success’ state in finite time. The progress requirement demands that at least one of those interactions in the ‘all-met’ state enter into the ‘success’ state. This requirement is satisfied in our algorithm. In order to prove this, we need the lemma as follows.

Lemma 2. If a thread $T_{j,ir}$ sends a *Withdraw* message to I_{ir} in the ‘request-sent’ state and enters into the ‘re-try’ state, there must be another thread $T_{j,ir}$ of P_j in the ‘commit-sent’ state such that $r' < r$.

Proof: A thread $T_{j,ir}$ in the ‘request-sent’ state sends a *Withdraw* message 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, the thread $T_{j,ir}$ is either the first thread in the ‘commit-sent’ state or a past pending thread which has been woken up by another thread and entered into the ‘commit-sent’ state.

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}, \dots, I_{uw} be the subset of all the interactions that enter into the ‘all-met’ state after receiving all the requests they need. Then, at least one of them will enter into the ‘success’ state.

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

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 the ‘pending’ state. This is because otherwise it must receive a *Stop* from M_{vj} and therefore there must be a thread $T_{vi,ui}$ ($1 \leq i' \leq w$) that succeeds in its coordination. This would imply that $I_{ui'}$ enters into the ‘success’ state.

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 the ‘meeting’ state, it must have received at least one *Withdraw* message from, say $T'_{j_1,w}$ ($1 \leq j_1 \leq y$). According to Lemma 2, there must be a thread T'_{j_1,i_1} that has sent a *Commit* message to I'_{i_1} such that $i_1 < w$. Since I'_{i_1} also enters into the ‘meeting’ state, it must have received a *Withdraw* message from say, T'_{j_2,i_1} ($1 \leq j_2 \leq y$).

By using Lemma 2 again, we can find another thread T'_{j_2,i_2} that has sent a *Commit* message 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.

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. By taking advantage of multi-threading supported by modern operating systems, this algorithm requires less messages than the algorithm proposed by Joung and Smolka [4]. In this algorithm, middleware serves as the underlying communication infrastructure. Data exchanges are done by middleware. Application developers can develop distributed applications without concern about the issues of heterogeneity such as data marshalling/unmarshalling and data formats. In addition, no specific language processor needs to be implemented in order to execute IP programs. Our model allows IP programs to be executed in any general programming environment. Finally, the concerns of heterogeneity, distribution, communication, and coordination are separated into a 4-layered interaction model. The model isolating computation, communication, and coordination promotes reuse, improves comprehension, and eases maintenance due to software evolution.

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)*, 2003, 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*, 1999, pp. 458-465.
- [3] N. Francez and I. R. Forman, *Interacting Processes*, Addison-Wesley, 1996.
- [4] Y.-J. Joung and S. Smolka, ‘Strong Interaction Fairness via Randomization,’ *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 2, February 1998, pp. 137-149.
- [5] M. Radestock and S. Eisenbach, ‘Component Coordination in Middleware Systems,’ *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, 1998, pp. 225-240.