Implementing A Distributed System Protocol :Paxos

Abstract

This project will discuss approaches to solving consensus problem in distributed computing and multi-agent system at first and then implement a protocol to solve consensus: the Paxos protocol. The consensus problem requires agreement among a number of processes (or agents) for a single data value. Some of the processes (agents) may fail or be unreliable in other ways, so consensus protocols must be fault tolerant or resilient. The processes must somehow put forth their candidate values, communicate with one another, and agree on a single consensus value. To solve consensus, there are different protocols: two-phase commit protocol, three-phase commit protocols and Paxos protocol. After analysis those approaches, I find that Paxos is the best approach and prove it can solve consensus problem by implement Paxos algorithm and use it to solve select master node problem.

Keyword: consensus two-phase commit protocol three-phase commit protocol Paxos select master node problem

1.Problem description

To achieve reliability in distributed system, we want the whole system will agree on only 1 value for a specific data fast enough to ensure system can move on although there are fault in the system. To process of reaching agreement is consensus and consensus protocol work as guider about what to do under different situation: what system should act if there are no error? what system should act if some node dead? what system should act if there are message lost for some node? The key point of consensus, just as the name suggest, is not about the what will be agreed, but what reach a same conclusion.

We often need to reach consensus in following situation:

a.synchronizing replicated state machines and making sure all replicas have the same (consistent) view of system state.

b.electing a leader (e.g., for mutual exclusion)

c.distributed, fault-tolerant logging with globally consistent sequencing

d.managing group membership

e. deciding to commit or abort for distributed transactions

2. important existing solution

There are some solutions to solve consensus: two-phase commit protocol, three-phase commit protocols and Paxos protocol. I will introduce those protocols at first and then compare those protocol.

a. two-phase commit protocol(2PC)

Two-phase commit protocol is widely used in database to deal with distributed transaction among multiple databases. It going through following phase to reach an agreement:

Phase	description	memo
Prepare phase	The initiating node, called the global coordinator, asks participating nodes other than the commit point site to promise to commit or roll back the transaction, even if there is a failure. If any node cannot prepare, the transaction is rolled back.	1 node act as coordinator and other node act as cohort.
Commit phase	If all participants respond to the coordinator that they are prepared, then the coordinator asks the commit point site to commit. After it commits, the coordinator asks all other nodes to commit the transaction.	It is like voting in daily life.
Forget phase	The global coordinator forgets about the transaction.	After forget, the coordinator is ready for next round.

There are few assumption about two phase commit protocol:

a. No node will crush forever.

b.If crush happens, the crushed can recover by using log.

c. The log is never lost or corrupted in a crush

d. any 2 nodes can communicate with each other.

The greatest disadvantage of the two-phase commit protocol is that it is a blocking protocol. If the coordinator fails permanently, some cohorts will never resolve their transactions: After a cohort has sent an agreement message to the coordinator, it will block until a commit or rollback is received.

b. three-phase commit protocol(3PC)

To deal with the drawbacks of two-phase commit protocol, three- phase commit protocol(3PC) occurs.3PC is non-blocking and specifically, 3PC places an upper bound on the amount of time required before a transaction either commits or aborts. This property ensures that if a given transaction is attempting to commit via 3PC and holds some resource locks, it will release the locks after the timeout. 3PC needs to go through following phases to reach an agreement:

Phase	description	memo
Prepare phase	If there is a failure, timeout, abort. The coordinator sends a canCommit? message to the cohorts and moves to the waiting state.The cohort receives a canCommit? message from the coordinator. If the cohort agrees it sends a Yes message to the coordinator and moves to the prepared state. Otherwise it sends a No message and aborts. If there is a failure, it moves to the abort state.	It will not blocking so coordinator can move to next agreement. It will communicate the outcome to everyone in the cohort.

Pre-commit phase	If there is a failure, timeout, or if the coordinator receives a No message in the waiting state, the coordinator aborts the transaction and sends an abort message to all cohorts. Otherwise the coordinator will receive Yes messages from all cohorts within the time window, so it sends preCommit messages to all cohorts and moves to the pre- prepared state. In the prepared state, if the cohort receives an abort message from the coordinator, fails, or times out waiting for a commit, it aborts. If the cohort receives a preCommit message, it sends an ACK message back and awaits a final commit or abort.	This stage solving the infinite waiting situation when coordinator crush.
Commit phase	If there is a failure, timeout in the waiting state, the coordinator aborts the transaction and sends an abort message to all cohorts. Otherwise the coordinator will receive ACK messages from all cohorts within the time window, so it sends doCommit messages to all cohorts and moves to the prepared state. In the commit state, if the cohort receives an abort message from the coordinator, fails, or times out waiting for a commit, it aborts. If the cohort receives a doCommit message, it sends an haveCommitted message back and awaits a final commit or abort. After receive haveCommitted , the coordinator asks all other nodes to commit the transaction.	It is like voting in daily life.

c. Paxos protocol

Another approach is Paxos protocol. Unlike the 2PC or 3PC, it have a more complicated role in this algorithm:

role	description
client	The Client issues a request to the distributed system, and waits for a response. For instance, a write request on a file in a distributed file server.
acceptor	The Acceptors act as the fault-tolerant "memory" of the protocol. Acceptors can form Quorums (cf. the definition of Quorum below). Any message sent to an Acceptor must be sent to a Quorum of Acceptors. Any message received from an Acceptor is ignored unless a copy is received from each Acceptor in a Quorum (e.g., Promises sent to a Proposer, or Accepted messages sent to a Learner).

Quorums	Quorums are defined as subsets of the set of Acceptors such that any two subsets (that is, any two Quorums) share at least one member.
proposer	A Proposer advocates a client request, attempting to convince the Acceptors to agree on it, and acting as a coordinator to move the protocol forward when conflicts occur.
learner	Learners act as the replication factor for the protocol. Once a Client request has been agreed on by the Acceptors, the Learner may take action (i.e.: execute the request and send a response to the client). To improve availability of processing, additional Learners can be added.
leader	Paxos requires a distinguished Proposer (called the leader) to make progress. Many processes may believe they are leaders, but the protocol only guarantees progress if one of them is eventually chosen. If two processes believe they are leaders, they may stall the protocol by continuously proposing conflicting updates. However, the safety properties are still preserved in that case.

Because Paxos protocol family is really complicated and we only introduced the basic Paxos (the idea propose in *“Paxos made simple”*).

Phase	description
1a: Prepare phase	A Proposer (the leader) creates a proposal identified with a number N. This number must be greater than any previous proposal number used by this Proposer. Then, it sends a Prepare message containing this proposal to a Quorum of Acceptors. The Proposer decides who is in the Quorum.
1b:promise phase	If the proposal's number N is higher than any previous proposal number received from any Proposer by the Acceptor, then the Acceptor must return a promise to ignore all future proposals having a number less than N. If the Acceptor accepted a proposal at some point in the past, it must include the previous proposal number and previous value in its response to the Proposer. Otherwise, the Acceptor can ignore the received proposal. It does not have to answer in this case for Paxos to work. However, for the sake of optimization, sending a denial (Nack) response would tell the Proposer that it can stop its attempt to create consensus with proposal N.

2a: Accept request	If a Proposer receives enough promises from a Quorum of Acceptors, it needs to set a value to its proposal. If any Acceptors had previously accepted any proposal, then they'll have sent their values to the Proposer, who now must set the value of its proposal to the value associated with the highest proposal number reported by the Acceptors. If none of the Acceptors had accepted a proposal up to this point, then the Proposer may choose any value for its proposal.[17] The Proposer sends an Accept Request message to a Quorum of Acceptors with the chosen value for its proposal.
2b: Accepted	If an Acceptor receives an Accept Request message for a proposal N, it must accept it if and only if it has not already promised to any prepare proposals having an identifier greater than N. In this case, it should register the corresponding value v and send an Accepted message to the Proposer and every Learner. Else, it can ignore the Accept Request.

d. comparison between between Paxos ,2PC and 3PC

Here is a message complexity between Paxos and 2PC.Because 3PC through more stage, so if it ends in the last stage, the communication cost is lager than 2PC. So if we can prove that Paxos is better than 2PC in cost, we can prove that Paxos is better than 3PC.

Character	2PC	Paxos
Message Delays	5	5
no co-location message	3N – 1	(N + 1)(F + 3) – 4
with co-location	3N – 3	N (F + 3) – 3
Writes to stable storage	N + 1	N + F + 1

N: number of nodes F: number of failure

If F=0, the message complexity of Paxos is equal with the message complexity of 2PC but have lower message Delay. What's more, Paxos is nonblocking protocol and 2PC is a blocking protocol. So Paxos is better for it has lower message delay then 3PC and its nonblocking character is better than blocking character for 2PC.

3. Description of my solution

In this part, I use Paxos to solve select master node problem and want to see wether it can solve consensus problem .

3.1 select master node problem

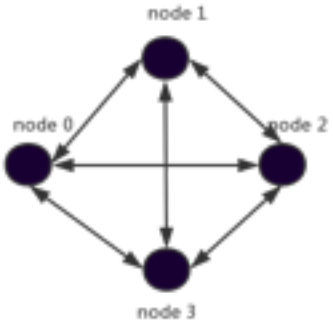
In a small network, each node want to have access to a specific resource(memo in each nodes). However, to achieve safe access to resource, we can only allow 1 node access to resource for a specific time and all changes on the resource will be learned by other nodes. So how to decided the node for each turn? In safe case, we can have an order to do this, which means no matter which node become the master node, all node will node the changes eventually. But when there is an error occur, how can all nodes know the changes?

3.2 How to use PAXOS algorithm to implement select master node in a network?

In the problem, each node can act like a proposer,acceptor and learner in PAXOS algorithm and their proposal is access to the resource. For PAXOS algorithm can only accept 1 proposal among multiple proposals each time, we can choose the node whose proposal is accepted as the master node.

Because each node is the proposer,acceptor and learner, the communication is N to N and the network topology may looks like this:

In example, it is a network with 4 nodes.



In select master node situation, the PAXOS set a conference between each node in the network for k round. Each round can let only 1 node be the master node and other node should learn that node's proposal(changes on memo). In each round, they can be proposer, acceptor or learner if no error occurs.In error case, the role is limited to acceptor or learner if message lost or learner if the node dead.

case	role's change	Description
NO_ERROR	proposer->acceptor->learner	In this case, a node can be proposer to propose a request, accept other's proposals and learn from it self or others.
NODE_DIE	learner	If a node die, because PAXOS protocol assumed that a message will eventually send to each member in the network, it will learn from other node but can't vote or promise for a proposal.
NODE_LOST_MESSAGE	acceptor, learner	If a node lost message for a specific round, it gives up the right to vote for this round and can only vote and learn from others.

There is a situation when all node die or lost message for a round, then there is no communication between each nodes. If a node is error-free and have k tasks to propose and there are p alive nodes in the network, so we need kp rounds to finish update.

I use MPICH to implement Paxos algorithm and use C to write my code. To select master node for each round and finish all update tasks, we need use following data structure.

data structure name	type	description	function

Proposal	struct Proposal{
	int tag;
	char var;
	char data
	}
		The content of a proposal proposed by a proposer
		example of a Proposal:
		tag=1
		var='a'
		data=20
			If a proposer's proposal satisfy:
			1. can communicate successfully with majority of acceptor
			2. uses a proposal with tag greater than proposer_number
			So if the proposal with tag greater than proposer_number, it will be the leader for a set of acceptor and other proposer will wait leader's proposal been "accepted" then they can "prepare" a proposal. Other situation is seen in

memo	struct memo{
	char var;
	int data;
	int N;
	int node_state
	}
		A memo kept by each acceptor, it will record previous proposal.	node_state can be die,safe or message lost. We can set state for each node and let it play different role following Paxos requirement.

acceptor_ state	int [];	An array to decided wether a legal proposal can be accepted.	In the beginning, for all acceptor, they are open to accept any proposal(if the proposal legal: tag>=N), once a proposer's proposal reach state "accept" on this acceptor, this acceptor can't accept any proposal. When that proposer's proposal has been accepted by all, this acceptor can promise new proposal.

task	struct task{
	char var;
	int value;
	};
		An struct to save prat of proposal	Each node with rank i only update 1 value for a specific variable. For example, node with rank=0 will only update value for a.

promise_time	double[]	An array to save time for receive message from majority of acceptor	The promised proposal with the minimum promise_time will be accepted.
promise_state[]	int []	Wether node's proposal has been passed by majority of accepter(or learner)	If Proposal i is promised by majority, promise_state[i]==1, otherwise, promise_state[i]==0.
p_queue	array in type of struct Proposal	used to save proposal send from all nodes	If a node is dead or lost message, it will send blank proposal. According to the assumption of Paxos, a node will eventually recover from error, so we assumed that it will be recover in the end of each round when an accepted proposal occurs.
task_num	integer	How much round will among those nodes.	When task_num =1, we can exam the correctness by check wether the number of carried out task is 1; When task_num >1, we can exam the correctness of Paxos with multiple proposal.

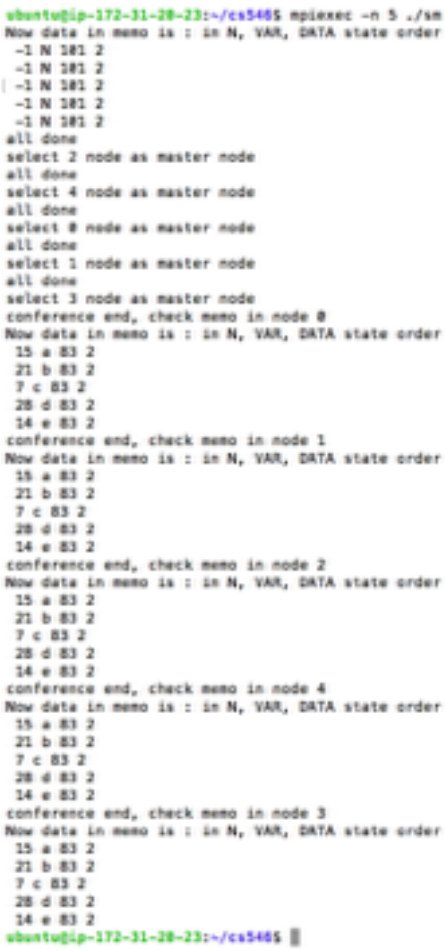
index	integer	Used to represent the node whose proposal has been accepted	To update local memo

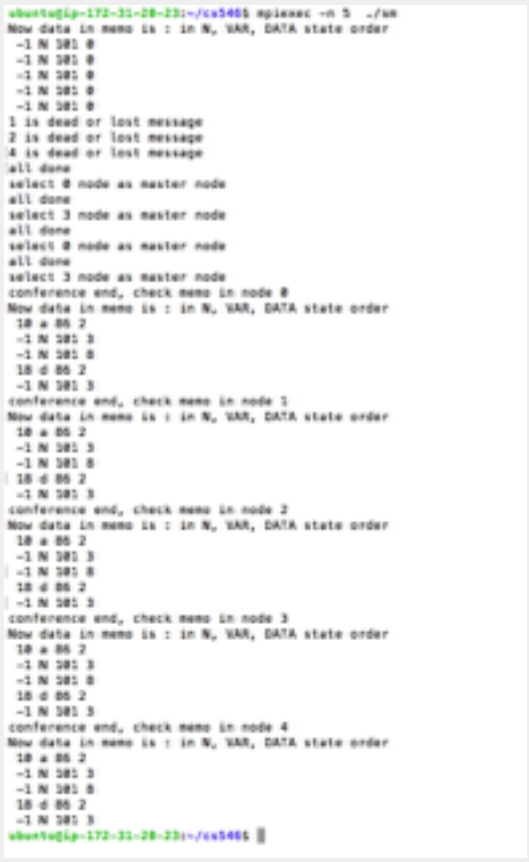
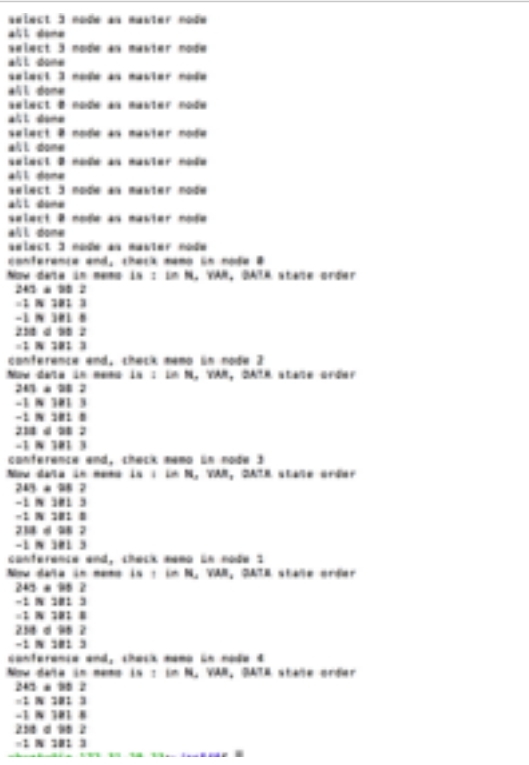
For each node, it will go through following stage to be a master node:

CASE NAME	Message flow
Initialization	node with rank i —->node with rank i: it will do following steps: set node state: safe, dead or lost message. set value for proposal. set acceptor_sate to 1: open to accept
Prepare	client—-> Proposer<——> All nodes in the system(As Acceptor or Proposer)
Promise_Success	Acceptor(all nodes)<—->Proposer(all nodes) : in this case, sand promise message
Promise_failed	Acceptor(all nodes)<—->Proposer(all nodes) : in this case, sand reject message
Accept	node with rank i —->node with rank i: Wether its' proposal being promised or not, update local memo to stay with the promised proposal.

4. result

Here are the results under for different test case.

test_case_ name	test aspest	result	state	memo
SAFE CASE: all node CAN ACT AS PROPOSER ,ACCEPTOR OR LEARNER				
Basic function: select master node	select only 1 node be master node for each round, update memo successfully .		pass	By observing the N in the memo, we will find that if the node is select as master more earlier , it will have small N: node with rank=2 select master node at the first round, so N=7. node 3 select as the master node in the last round so it have the biggest N and we know that the proposal for node 3 has been declined for 4 times.
stress for more round	To see wether the program will crush when in more round		pass	let tasknum=20, so we need 100 round
ERROR IN NETWORK: some node can lost or dead				

Run for 4 round	test: when the program end, all memo should be same. Lost or dead node can't be master node		pass	
stress for more round	test: to see wether the program will crush		pass	

5. conclusion

From the result part, we can see that the Paxos algorithm works fine if there are dead node and can fulfill our requirement: reach consensus for a network. The reason about why I choose MPICH is that Paxos algorithm works for massage passing platform. If it is shared memory system, we don't need to do this: because if error occurs in 1 node, it won't affect the system, it can fetch data from shared memory and if the write operation is abort because the error, it only need to read and write again.

About MPICH and parallel programming, MPICH is a really powerful message passing interface but we should be careful while using it. The first thing is tag for communication. In MPICH, tag is used to identify a message send or receiver by sender of receiver. So sender and receiver changed, make sure the pair of sender-receiver use same tag. Then is the non-blocking communication. Af first, I use blocking communication, which leads to dead lock problem, then I changed it to non-blocking communication. Although the dead lock problem solved, the program say hanging due to the mis-use of tag. So I change it to MPI_Bcast. The third thing is the importance of good design. I used to do my homework with out detailed design, but for this project, it always forced me to figure out a better way(or you will waste lost of time in debug). So before coding, I should figure out the communication between different nodes. This idea is different with Pthread: coding using Pthread is more straight forward, just ask for privilege for mutex and with for condition then access the resource you want. However in MPICH, it is more about dealing relationship with multiple nodes instead of tasks. In parallel programming, data race may occur. In former homework, I avoid it successfully because master node won't change. However, in this project, I trapped by this problem for many times because each node will send and receive from all node. To avoid this situation, we need use different variable(or register) to store the data.

For PAXOS algorithm, it is inserting to see how to stay consistency between different node. In basic case, for each node want to be the master node, and only 1 node can be the master node and let acceptor accept it's value, so PAXOS set following rule to achieve this: each proposal have a unique tag: N, if s new proposal was proposed, all member should pass it. If the proposer think his proposal is new and propose it(for example, set a =10), when it send this proposal to acceptor, acceptor may have an accepted proposal(set a=5), so this proposal is rejected and the proposer should propose it again in the next round. Before promise, all proposer can propose proposal so if it is not a new proposal, proposal with bigger N will be accepted. In unusual case, if a node die or lost message, it can still be the learner and receive data.

6. Reference

[1]Lamport, Leslie (May 1998). "The Part-Time Parliament". ACM Transactions on Computer Systems.

[2]Lamport, Leslie (2001). Paxos Made Simple ACM SIGACT News (Distributed Computing Column)

[3]Oracle Database Administrator's Guide from Oracle website

[4]Jim Gray and Leslie Lamport(2004)Consensus on Transaction Commit

[5] Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman (1987): Concurrency Control and Recovery in Database Systems, Chapter 7

[6] Gerhard Weikum, Gottfried Vossen (2001): Transactional Information Systems, Chapter 19

[7]Skeen, Dale; Stonebraker, M. (May 1983). "A Formal Model of Crash Recovery in a Distributed System"

7. Appendix

7.1 source code

See Select_master_node.c

7.2 screenshot for debug stage

```
===================================================================================
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   PID 31389 RUNNING AT ip-172-31-20-23
=   EXIT CODE: 139
=   CLEANING UP REMAINING PROCESSES
=   YOU CAN IGNORE THE BELOW CLEANUP MESSAGES
===================================================================================
YOUR APPLICATION TERMINATED WITH THE EXIT STRING: Segmentation fault (signal 11)
This typically refers to a problem with your application.
Please see the FAQ page for debugging suggestions
```

Segment fault for MPI_Bcast

```
Fatal error in PMPI_Barrier: Message truncated, error stack:
PMPI_Barrier(425)...................: MPI_Barrier(MPI_COMM_WORLD) failed
MPIR_Barrier_impl(332)..............: Failure during collective
MPIR_Barrier_impl(327)..............:
MPIR_Barrier(292)...................:
MPIR_Barrier_intra(150).............:
barrier_smp_intra(111)..............:
MPIR_Bcast_impl(1452)...............:
MPIR_Bcast(1476)....................:
MPIR_Bcast_intra(1287)..............:
MPIR_Bcast_binomial(239)............:
MPIC_Recv(353)......................:
MPIDI_CH3U_Request_unpack_uebuf(568): Message truncated; 4 bytes received but bu
ffer size is 1
```

abort for MPI_Bcast-a :print error code

```
-1 N 101
-1 N 101
Finish prepare stage in 1
start promise stage in 1
Finish communicate

Program received signal SIGSEGV, Segmentation fault.
0x0000000000401c93 in Promise_Accept_Accepted (p=..., myrank=1, m=0x604080,
    acceptor_state=0x603110 <acceptor_state>) at Select_matster_node.c:296
296                     m[send_rank].data=send_data;
(gdb) p send_rank
$1 = 1684234849
(gdb)
```

abort for MPI_Bcast-b :in GDB

```
ubuntu@ip-172-31-20-23:~/cs546$ mpiexec -n 5 valgrind -q ./sm
cr_libinit.c:189 cri_init: sigaction() failed: Invalid argument
cr_libinit.c:189 cri_init: sigaction() failed: Invalid argument
cr_libinit.c:189 cri_init: sigaction() failed: Invalid argument
cr_libinit.c:189 cri_init: sigaction() failed: Invalid argument
cr_libinit.c:189 cri_init: sigaction() failed: Invalid argument

=====================================================================================
===
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   PID 1193 RUNNING AT ip-172-31-20-23
=   EXIT CODE: 134
=   CLEANING UP REMAINING PROCESSES
=   YOU CAN IGNORE THE BELOW CLEANUP MESSAGES
=====================================================================================
===
YOUR APPLICATION TERMINATED WITH THE EXIT STRING: Aborted (signal 6)
This typically refers to a problem with your application.
Please see the FAQ page for debugging suggestions
```

system bug

7.3 Debug

7.3.1 debug tool

Using xterm+gdb to detect common bug and Using valgrind to detect memory leak.

7.3.2 log

7.3.2.1 check passing correct parameters for each function

function name	parameters	state	solve problem	cause
Inittask_set	char* task_set,struct memo m[],int acceptor_state[]	pass	2 Segment fault	type miss match char or char* …
get_proposer_number	int taskth,int myrank	pass		
Prepare	int N,int myrank	pass		
Promise_Accept_Accepted	struct Proposal p, int myrank,struct memo m[],int acceptor_state[]			

promise_stage	int data, int rec_buf_for_data[]	pass		

7.3.2.2 eliminated bugs

error name	description	solution
dead lock	rank 0 rank 1 MPI_RECV(tag0) MPI_Send(tag0) MPI_Send(tag1) MPI_RECV(tag0) MPI_RECV(tag0) may keep waiting if in rank 1 MPI_Send(tag1) send first.	By analysis message flow, replace multiple send-receive pair to broadcast or gather(because in this problem, it is an N to N communication)
MPI_Bcast Hanging	if(rank== i) MPI_Bcast(…) MPI_Bcast shouldn't in any if(rank== …) statement; and error in root for detail,see in screenshot	check root, need set buffer for it. At first, I use a struct array to broadcast, but failed. For although we have malloc space for it, we didn't finish initialization, so we need to set buffer and use data from broadcasting to initialize this element in struct array.
segment fault	if(statement unrelated with rank) MPI_Bcast(…) For each MPI_Bcast, it should run on all node for detail,see in screenshot	discard unnecessary communication in accept stage
abort	abort for detail,see in screenshot	don't ruin the result but is a bug for Ubuntu system running MPICH with version higher than 2.0 when using valgrind.