# Final Project Report

105062206 蔡哲維
105062306 胡世昕

**Implementation:**

Client side:

```
    public RemoteTerminalEmulator(StatisticMgr statMgr) {
        this.statMgr = statMgr;

        // Set the thread name
        //setName("RTE-" + rteCount.getAndIncrement());
        this.selfId = App.clientId;
        this.targetServerId = selfId % serverCount;
        this.client = new VanillaCommClient(this.selfId, this);
        new Thread(client).start();
    }
```

在 RTE constructor 中，建立一個 VanillaCommClient 來和 server 端溝通。

```
protected void executeTxn(VanillaCommClient client, Object[] pars, int targetServerId) throws SQLException {
    /*switch (BenchmarkerParameters.CONNECTION_MODE) {
    case JDBC:
        Connection jdbcConn = conn.toJdbcConnection();
        jdbcConn.setAutoCommit(false);
        result = getJdbcExecutor().execute(jdbcConn, pg.getTxnType(), pars);
        break;
    case SP:
        result = conn.callStoredProc(pg.getTxnType().getProcedureId(), pars);
        break;
    }*/
    client.sendP2pMessage(ProcessType.SERVER, targetServerId, pars);
}
```

當 parameter generator 生成出 parameter 後，呼叫 executeTxn，將資料傳給 server 端。

```
public void onReceiveP2pMessage(ProcessType senderType, int senderId, Serializable message) {
    //System.out.println(message);
    SpResultSet rs = (SpResultSet) message;
    VanillaDbSpResultSet result = new VanillaDbSpResultSet(rs);
    this.txnEndTime = System.nanoTime();
    txnRT = txnEndTime - txnRT;
    System.out.println("MicroBenchMark: " + result.outputMsg());
    TxnResultSet final_result = new TxnResultSet(MicrobenchTransactionType.MICRO_TXN, txnRT, txnEndTime,
            result.isCommitted(), result.outputMsg());
    if (!isWarmingUp)
        statMgr.processTxnResult(final_result);
    synchronized(this) {
        this.notify();
    }

}
```

一旦 VanillaCommClient 接收到 server 端回傳的 resultSet，開始進行包裝，並將該資料傳入 statMgr 中。

Server side:

```
public class CalvinServerStartUp {
    public static void main(String[] args) {
        GroupCommModule.startGroupComm(Integer.parseInt(args[0]));
        SutStartUp sut = null;
        sut = new VanillaDbSpStartUp();
        if (sut != null)
            sut.startup(args);

    }

}
```

CalvinServerStartUp 的 main 為 server 端的程式進入點，在此做了兩件事，一是去啟動 GroupCommModule，另外一件事是去 init VanillaDB。

```
public static void startGroupComm(int selfId) {
    if (logger.isLoggable(Level.INFO))
        logger.info("Initializing the Group Communication Module...");
    moduleId = selfId;
    groupCommServer = new VanillaCommServer(selfId, new GroupCommModule());
    new Thread(groupCommServer).start();
    createClientRequestHandler();
    //clientHandler();


}
```

GroupCommModule 中的 startGroupComm 用來建立一個 VanillaCommServer 物件，用來進行 server 間以及 server-client 間的資訊傳遞。

```
private static void createClientRequestHandler() {
    new Thread(new Runnable() {

        @Override
        public void run() {
            epochStart = System.currentTimeMillis();
            while (true) {
                try {
                    if(System.currentTimeMillis() - epochStart >= 10 && !msgQueue.isEmpty()) {
                        messages = new ArrayList<Serializable>();
                        while(!msgQueue.isEmpty()) {
                            messages.add(msgQueue.take());
                        }
                        groupCommServer.sendTotalOrderMessages(messages);
                        epochStart = System.currentTimeMillis();
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }

    }).start();
}
```

此 method 負責每 10ms，將該 GroupCommModule 所蒐集到 client request，經由 totalordering 的方式，傳給其他 partition 的 VanillaCommServer 物件。

```
public class Scheduler {
    private static Executor executor = Executors.newFixedThreadPool(50);
    private static HashMap<Long, Integer> MetaData = new HashMap<Long, Integer>();
    public static CopyOnWriteArrayList<SpResultSet> Cache = new CopyOnWriteArrayList<SpResultSet>();
    private static int serverCount = ProcessView.buildServersProcessList(-1).getSize();
    static class TpccWork implements Runnable{
        NewOrderProc sp;
        public TpccWork(NewOrderProc sp) {
            this.sp = sp;
        }
```

此為 Scheduler。其主要工作為分析接收到的 totalordering message、新建 storedprocedure、利用 conservative locking 的方式取得該當前 txn 所需要的鎖、最後是將執行 storedprocedure 的工作交個 thread pool 中的 working thread。

```
  public static void analyzeTheMicroMessage(Serializable message) {
      //Analyze the serialized message
      Object[] deserialized = (Object[]) message;
      for(int i=0;i<deserialized.length;) {
          Object[] pars = new Object[33];
          for(int j=0;j<33;j++) {
              pars[j] = deserialized[i];
              i++;
```

此 method 是用來處理 Microbenchmark 的 message（由於時間的關係，現在只能使用特定版本的 microbenchmark）。

```
//conservative locking
private static void conservativeLocking(MicroTxnProc sp) {
    //Conservative locking
    for (int idx = 0; idx < sp.getParamHelper().getReadCount(); idx++) {
        int iid = sp.getParamHelper().getReadItemId(idx);
        Scan s = StoredProcedureHelper.executeQuery(
            "SELECT i_name, i_price FROM item WHERE i_id = " + iid,
            sp.getTransaction()
        );
        s.beforeFirst();
        if (s.next()) {
            s.getSlock();
            s.getRecordXlock();
        }
        s.close();
    }

}
```

此 method 是用 conservative 的方式取得該 txn 所需要的 lock。

```java
static class MicroWork implements Runnable{
    MicroTxnProc sp;
    public MicroWork(MicroTxnProc sp) {
        this.sp = sp;
    }

    @Override
    public void run() {
        SpResultSet rs = sp.execute();
        SpResultRecord record = (SpResultRecord)rs.getRecord();
        //System.out.println(record.getFldValueMapSize());
        //System.out.println(MetaData.get(sp.getTransaction().getTransact
        if(record.getFldValueMapSize() == 21) {
            sp.getTransaction().commit();
            rs.setCommitted();
            int clientId = sp.getParamHelper().getClientId();
            GroupCommModule.groupCommServer.sendP2pMessage(ProcessType.CL
        }
        else if(MetaData.get(sp.getTransaction().getTransactionNumber())!
            //System.out.println("Here");
            rs.setTxNum(sp.getTransaction().getTransactionNumber());
            sp.getTransaction().commit();
            rs.setCommitted();
            GroupCommModule.groupCommServer.sendP2pMessage(ProcessType.SE
            MetaData.remove(sp.getTransaction().getTransactionNumber());
        }
```

此 class 為 working thread 所要執行的工作。在執行完 storedprocedure 的工作後，working thread 需要根據目前有哪些 partition 需要自己的執行結果，用 p2p 的方式傳給該 partition 的 communication module。

**Experiment:**

由於我們目前做到的功能只能 handle 我們實驗時跑的設定，故若想要重複我的們實驗需要使用我們 git 上的 DB file，以及設定檔。

環境：

實驗參數設定：

```
# The number of remote terminal executors for benchmarking
org.vanilladb.bench.BenchmarkerParameters.NUM_RTES=1
# The IP of the target database server
org.vanilladb.bench.BenchmarkerParameters.SERVER_IP=127.0.0.1
# 1 = JDBC, 2 = Stored Procedures
org.vanilladb.bench.BenchmarkerParameters.CONNECTION_MODE=2
# 1 = Micro, 2 = TPC-C,
org.vanilladb.bench.BenchmarkerParameters.BENCH_TYPE=1
```

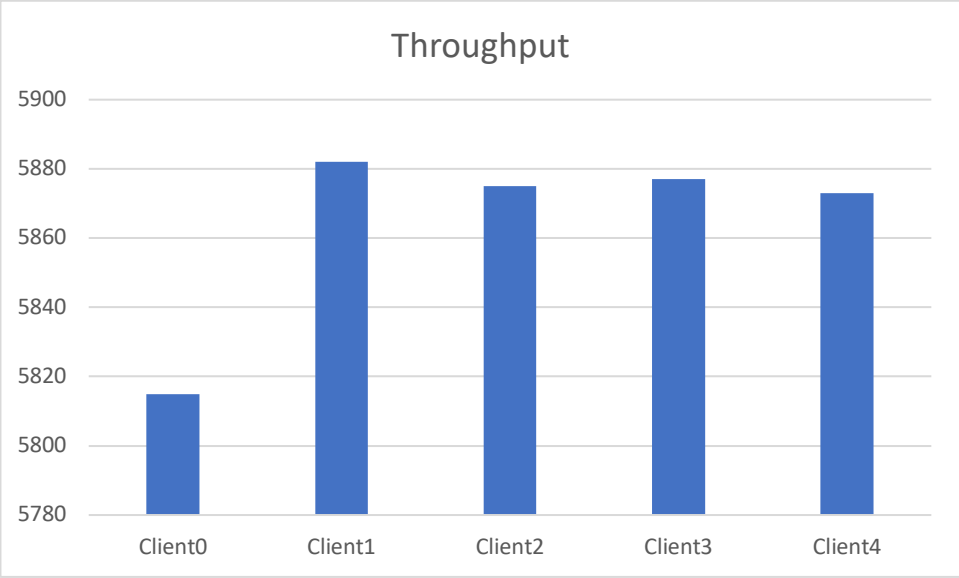
```
#
# Micro-benchmarks Parameters
#

# The number of items in the testing data set
org.vanilladb.bench.benchmarks.micro.MicrobenchConstants.NUM_ITEMS=300000
# The ratio of read-write transactions during benchmarking
org.vanilladb.bench.benchmarks.micro.rte.MicrobenchmarkParamGen.RW_TX_RATE=1.0
# The ratio of long-read transactions during benchmarking
org.vanilladb.bench.benchmarks.micro.rte.MicrobenchmarkParamGen.LONG_READ_TX_RATE=0.0
# The number of read records in a transaction
org.vanilladb.bench.benchmarks.micro.rte.MicrobenchmarkParamGen.TOTAL_READ_COUNT=10
# The number of hot record in the read set of a transaction
org.vanilladb.bench.benchmarks.micro.rte.MicrobenchmarkParamGen.LOCAL_HOT_COUNT=1
# The ratio of writes to the total reads of a transaction
org.vanilladb.bench.benchmarks.micro.rte.MicrobenchmarkParamGen.WRITE_RATIO_IN_RW_TX=1.0
# The conflict rate of a hot record
org.vanilladb.bench.benchmarks.micro.rte.MicrobenchmarkParamGen.HOT_CONFLICT_RATE=0.001
```

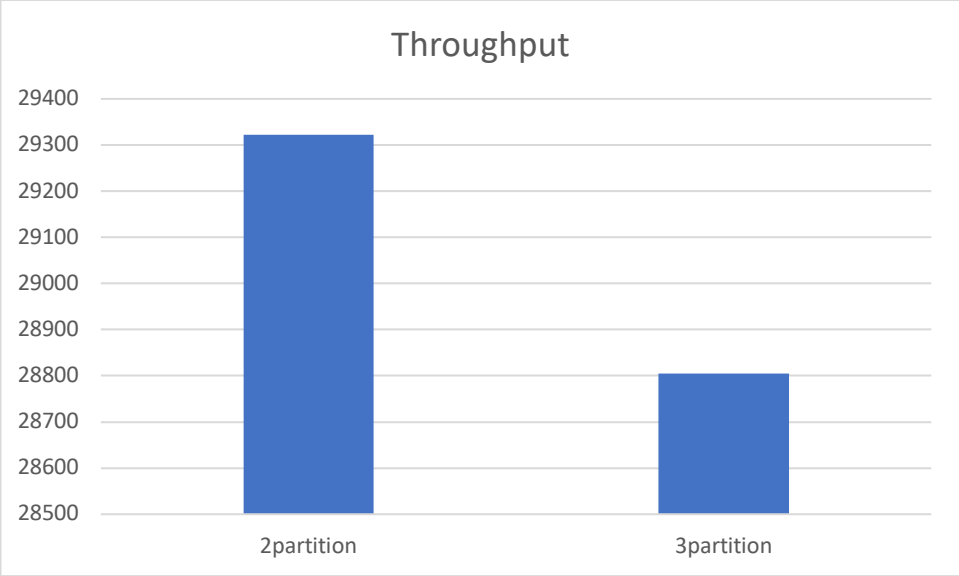
我們跑的實驗是使用五個 client process，每個 process 有一個 RTE。
目前跑的實驗有 2 partition 和 3p artition。

3partition:

2partition:




Compare:




Discussion:

照理來說，以 Calvin 的設計架構來看，3partition 的 throughput 應該比 2partition 的來得高，但是實驗結果卻不然。我認為這和我們的實作以及實驗環境有關。首先，因為實驗的機器只使用一台電腦，在核心數固定的情況下，多 partition 的反而不利，因為會有更多 thread 要執行。改善這問題的方式是使用多台電腦來開 server。我們在實驗過程中有試著進行多台機器的連線，但因為 IP 設定問題，沒辦法做到，目前還找不出原因何在。

另外因為多個 partition 之間的溝通傳遞也需要時間，當越多 partition 需要溝通

時，此傳遞時間所造成的 cost 就越明顯。
實驗的 client 數量太少也可能是一個原因，此原因導致 3-partition 的優勢無法展現出來。
而實作的問題主要在於，目前是讓多個 partition 的執行結果最後合併起來，傳到一個 partition，再由其負責傳回給 client。這會造成如果某 partition 並不需要其他 partition 的資料，但其他人都傳給他，再透過他來傳回給 server。這會比每個 partition 都各自個自己的 part 回傳給 client 多耗費一次的連線 cost。