Performance Benchmark of Corda Blockchain

Nihhaar Chandra R
nihhaar@gmail.com
May, 2019

Department of Computer Science & Engineering
IIT Bombay

Contents

Overview

Corda is an open-source distributed ledger technology for recording and processing financial agreements between regulated financial institute. In the world of various blockchain frameworks, Corda is an emerging permissioned blockchain framework designed for business from the start.

In this project, we aim to benchmark the performance of Corda. We also provide tips on how to capture various metrics relating to performance of Corda and the tools used in the benchmark in detail.

This work assumes you already have a basic idea of how Corda blockchain works and makes distributed ledger updates. The paper is organized as follows:

Chapter 1 discusses about some of the Corda internals which are not trivial from the docs. The theory presented here is used later on in the benchmark and in explaining the results.

Chapter 2 discusses about the concurrency in the enterprise version of Corda framework.

Chapter 3 talks about the metrics & tools used in benchmarking Corda, experimental design and setup, and the final results of the benchmark.

Chapter 4 discusses about the future scope of the project and provides many extensions to this project.

Chapter 1

Corda Internals

1.1 Introduction

We will be discussing about internals of the Corda, especially the messaging subsystem and concurrency in Corda. We will provide links to the relevant code in Corda whenever necessary for a good understanding of the framework.

1.2 Messaging in Corda

Corda [1] uses AMQP/1.0 over TLS between nodes which is currently implemented using Apache Artemis, an embeddable message queue broker.

A message broker acts as an intermediary platform when it comes to processing communication between two applications.

Apache ActiveMQ Artemis is an asynchronous messaging system, an example of Message Oriented Middleware. Queues are central to the implementation of the asynchronous interaction model within MOM as shown in figure 1.1. Message queues provide buffers that enables tasks to post and receive messages.

AMQP is a specification for interoperable messaging. It also defines a wire format, so any AMQP client can work with any messaging system that supports AMQP.

Brokers in Corda:

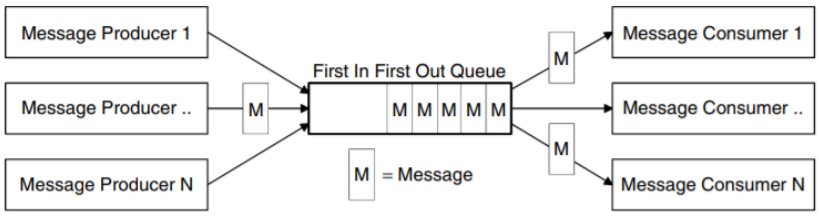

Figure 1.1: Producer-Consumer messaging pattern

1. The Artemis P2P broker
   - Handles P2P communication
   - Currently runs inside the node's JVM process, but in the future it will be able to run as a separate server
2. The Artemis RPC broker
   - Handles RPC communication
   - Currently runs inside the node's JVM process, but in the future it will be able to run as a separate server

Message Queues in Corda:

- p2p.inbound.$identity
  - The node listens for messages sent from other peer nodes on this queue.
  - Messages which are routed internally are also sent to this queue.
- internal.peers.$identity
  - Uses to route messages destined to other peers.
  - The broker creates a bridge from this queue to the peer's p2p.inbound.$identity queue, using the network map service to lookup the peer's network address.
- internal.services.$identity
  - Private queues the node may use to route messages to services.
- rpc.server
  - RPC clients send their requests here.

- rpc.client.$user.$random
  - Queue that is created on client side.
  - RPC requests are required to include a random number ($random) from which the node is able to construct the queue the user is listening on and send the response to that.

1.2.1 Network Map Service

Network map service is responsible for tracking public nodes on the network. Network map is needed by the nodes to learn the address of the other nodes for communication between them.

Nodes have an internal component, the network map cache, which contains a copy of the network map. When a node starts up, it submits its signed NodeInfo to the map service and fetches a copy of the full network map and caches it.

During development, it is convenient to skip deploying a network map server and just directly place the NodeInfo's in each node's cache directory.

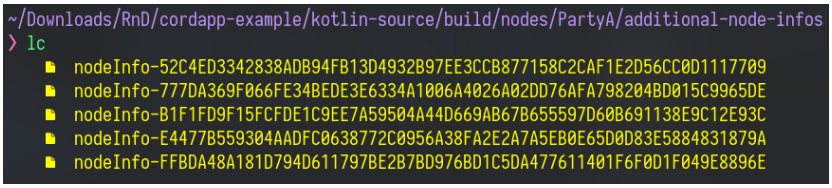

Figure 1.2: Node info's in the network map cache of PartyA node

1.2.2 RPC

RPC is the only way to interact with a Corda node. It allows you to connect to your node via a message queue protocol. You make calls on a JVM object as normal, and the marshalling back-and-forth is handled for you. Even the corda-webserver provided with the Corda uses RPC internally to expose the pre-defined actions over HTTP.

Protocol

In a high-level point of view, when a method is called on the interface the arguments are serialised and the request is forwarded to the server. The server then executes the code that implements the RPC and sends a reply.

The server consumes the queue "rpc.server" and receives RPC requests on it. When a client starts up it should create a queue for its inbound messages, this should be of the form "rpc.client.*username*.nonce". Each RPC request from client to server contains this address (queue name), this is where the server will send the reply to the request as well as subsequent Observations rooted in the RPC. The requests/replies are muxed using a unique rpc-request-id generated by the client for each request.

Observables The RPC system handles observables in a special way [2]. If an RPC reply payload from server to client contains observables then the server will generate a unique observable id for each observable and serialise them in place of the observables themselves. Subsequently an observable is created on the client to match the one on the server. Objects emitted by the server-side observable are pushed onto a queue which is then drained by the client. Detailed protocol is shown in figure 1.3.

```
An example session:
Client                          Server
  ----------RpcRequest(RID0)----------->   // Client makes RPC request with ID "RID0"
  <----RpcReply(RID0, Payload(OID0))----   // Server sends reply containing an observable with ID "OID0"
  <---------Observation(OID0)-----------   // Server sends observation onto "OID0"
  <---Observation(OID0, Payload(OID1))--   // Server sends another observation, this time containing another observable
  <---------Observation(OID1)-----------   // Observation onto new "OID1"
  <---------Observation(OID0)-----------
  -----ObservablesClosed(OID0, OID1)--->   // Client indicates it stopped consuming the observables.
  <---------Observation(OID1)-----------   // Observation was already in-flight before the previous message was processed
               (FIN)
```

Figure 1.3: RPC protocol in transferring a observable

For observables, the server must queue up objects emitted by the server-side observable until you download them. Note that the server side observation buffer is bounded, once it fills up the client is considered slow and will be disconnected.

You are expected to subscribe to all the observables returned, otherwise client-side memory starts filling up as observations come in. If you don't want an observable then subscribe then unsubscribe immediately to clear the client-side buffers and to stop the server from streaming.

RPC Client An RPC client connects to the specified server and allows you to make calls to the server that perform various useful tasks. RPC Client class runs on the client JVM.

`CordaRPCClient` class has a start method that takes the node's RPC address and returns a `CordaRPCConnection` object containing a proxy (`CordaRPCOps` object) that lets you invoke RPCs on the server. Calls on it block, and if the server throws an exception then it will be rethrown on the client. Proxy

translates API calls to lower-level RPC protocol messages.

```
final CordaRPCClient client = new CordaRPCClient(nodeAddress);
final CordaRPCConnection connection = client.start(username, password);
final CordaRPCOps rpcOps = connection.getProxy();
```

- `CordaRPCConnection` class is essentially just a wrapper for an `RPCConnection<CordaRPCOps>` and can be treated identically.
- The `CordaRPCOps` defines what client RPCs are available. It is implemented by `CordaRPCOpsImpl` class.
- The `RPCServer` implements the complement of `RPCClient`. When an RPC request arrives it dispatches to the corresponding function in `ops`.

Whenever start() method is called on `RPCConnection` object, a new session id is generated, a `RPCClientProxyHandler` object is created upon which start() is called thereby creating the client queue & starts the consumer session and the reaper.

```
val sessionId = Trace.SessionId.newInstance()
val proxyHandler = RPCClientProxyHandler(...)
proxyHandler.start()
```

1.2.3 Concurrency in Messaging

On server side The number of threads handling RPC requests - this defines how many RPC requests can be handled in parallel without queueing - is set to 4 in open source Corda.

The primary work done by the server thread is execution of flow logics, and related serialisation/deserialisation work [3]. All RPC calls execution takes place on the server thread, i.e, serially since there is only single server thread. That means if the server thread becomes too slow and a backlog of work starts to builds up it propagates back through into the messaging layer, which can then react to the backpressure. Artemis MQ in particular knows how to do flow control by paging messages to disk rather than letting the node run out of RAM [4].

On client side [5] Client RPC proxy is thread safe, blocking, and allows multiple RPCs to be in flight at once. Any observables that are returned and you subscribe to will have objects emitted in order on a background thread

pool. Each Observable stream is tied to a single thread, however note that two separate Observables may invoke their respective callbacks on different threads.

1.3 Flows in Corda

In Corda all communication takes the form of structured sequences of messages passed between parties which we call flows. Flows enable complex multi-step, multi-party business interactions to be modelled as blocking code without a central controller. The code is transformed into an asynchronous state machine, with checkpoints written to the node's backing database when messages are sent and received.

The flow abstracts all the networking, I/O and concurrency issues away from the node owner.

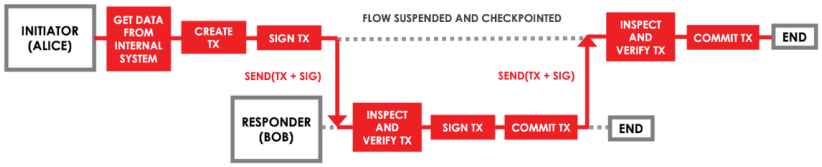

Figure 1.4: Lifecycle of a flow in Corda

Some of the benefits of flows:

- Avoiding "callback hell" in which code that should ideally be sequential is turned into an unreadable mess due to the desire to avoid using up a thread for every flow instantiation.
- Surviving node shutdowns/restarts that may occur in the middle of the flow by persisting to disk.
- Serialization.
- Message routing.

1.3.1 Quasar Fibers

A continuation is a suspended stack frame stored in a regular object that can be passed around, serialized, unserialized and resumed from where it was suspended. This concept is sometimes referred to as "fibers". JVM doesn't support continuation natively, so we use Quasar library.

Benefits of continuation:

- It allows us to write code that is free of callbacks, that looks like ordinary sequential code.
- A suspended continuation takes far less memory than a suspended thread.
- It frees the developer from thinking (much) about persistence and serialisation.

A flow is a class with a single call method. The call method and any others it invokes are rewritten by a bytecode rewriting engine called Quasar, to ensure the code can be suspended and resumed at any point. A flow is a subclass of `FlowLogic` implemented using direct, straight line blocking code.

1.3.2 State Machine

A state machine is a piece of code that moves through various states that indicate different stages in the progression of a multi-stage flow. Basically, a state machine manages a flow.

State machine is defined as a interface in `FlowStateMachine.kt` and implemented in `FlowStateMachineImpl.kt`.

1.3.3 State Machine Manager

A StateMachineManager is responsible for coordination and persistence of multiple `FlowStateMachine` objects.Each such object represents an instantiation of a (two-party) flow that has reached a particular point.

An implementation of this interface will persist state machines to long term storage so they can survive process restarts and, if run with a single-threaded executor, will ensure no two state machines run concurrently with each other.

State machine manager is defined as a interface in `StateMachineManager.kt` and implemented in `StateMachineManagerImpl.kt`.

1.3.4 Concurrency in Flow State Machine

The flow state machine is single threaded in open-source Corda. That means the number of flows that can run in parallel doing something and/or holding

resources like database connections is only **1** [6]. Note that at any point of time there may be many number of flows in suspended state, but only one flow can be run at a time holding resources.

1.4 Where to log various events in Corda?

This section contains where exactly in code a particular event starts or ends so that we can log the event start time, duration, etc...

1.4.1 RPC Connection

Start This event describes the connection establishment of a new rpc connection. On the client side, the connection to server is established in this function along with the session id and client queue name:

`https://github.com/corda/corda/blob/master/client/rpc/src/main/kotlin/net/corda/client/rpc/internal/RPCClient.kt#L64`

End The closing of rpc connection is also implemented in the same function thats starts the rpc connections:

`https://github.com/corda/corda/blob/master/client/rpc/src/main/kotlin/net/corda/client/rpc/internal/RPCClient.kt#L116`

1.4.2 RPC Flow Request

Start When we make start flow request in Corda, the first function invoked on the server side is (In general, any request from rpc is landed in `CordaRPCOpsImpl.kt`):

`https://github.com/corda/corda/blob/master/node/src/main/kotlin/net/corda/node/internal/CordaRPCOpsImpl.kt#L186`

End Flow is completed at the end of the function call in:

`https://github.com/Nihhaar/cordapp-example/blob/master/kotlin-source/src/main/kotlin/com/example/flow/ExampleFlow.kt#L73`

But in general, client calls get() on the `CordaFuture` object returned by `startFlowDynamic()` function to wait for the flow to complete. This is implemented in:

https://github.com/corda/corda/blob/master/core/src/main/kotlin/net/corda/core/internal/concurrent/CordaFutureImpl.kt#L172

1.4.3 Flow Events

Many events are in a flow like generating transaction, verifying transaction, signing transaction, notarization, etc... are present in the cordapp flow itself:

https://github.com/Nihhaar/cordapp-example/blob/master/kotlin-source/src/main/kotlin/com/example/flow/ExampleFlow.kt#L73

These events can be used to calculate metrics like consensus latency.

1.4.4 Tracking a Flow

If there are many flows running at an instant, it is hard to differentiate which flow event timestamp corresponds to which flow. So we need to also log a flow id that is unique to each flow. Fortunately, there is an inbuilt implementation for this. This unique flow id is called "StateMachineRunID".

(Defined in state machine here)
https://github.com/corda/corda/blob/release-V3.X/node/src/main/kotlin/net/corda/node/services/statemachine/FlowStateMachineImpl.kt#L44

(Defined in flow logic here as "runId")
https://github.com/corda/corda/blob/master/core/src/main/kotlin/net/corda/core/flows/FlowLogic.kt#L109

Chapter 2

Concurrency in Enterprise Corda

Corda Enterprise is a commercial distribution of Corda, a open source blockchain platform, specifically optimized to meet the demands of modern day business.

Considerable work has been done on Enterprise Corda to increase performance and to scale according to the resources available. The most important improvement in Enterprise version is multi-threaded state machine. This allows the flows to run in parallel to reap the benefits of multiple CPU cores and interleaving computation and I/O waits [7]. Benefit of multiple cores in enterprise corda is shown in figure 2.1.

`flowThreadPoolSize` is a configuration parameter in enterprise Corda that determines the number of threads available to handle flows in parallel. This is the number of flows that can run in parallel doing something and/or holding resources like database connections. A larger number of flows can be suspended, e.g. waiting for reply from a counterparty. When a response arrives, a suspended flow will be woken up if there are any available threads in the thread pool. Otherwise, a currently active flow must be finished or suspended before the suspended flow can be woken up to handle the event.

The default value of flow thread pool is two times the number of cores but can be configured depending on the hardware. Every thread will open a database connection, so for n threads, the database system must have at least n+1 connections available. Also, the database must be able to actually cope with the level of parallelism to make the number of threads worthwhile.

Enterprise Corda can reach the performance upto 1000 TPS [6] in terms of

throughput while open source version struggles to cross 10 TPS which we will see later on as shown in figure 2.1.

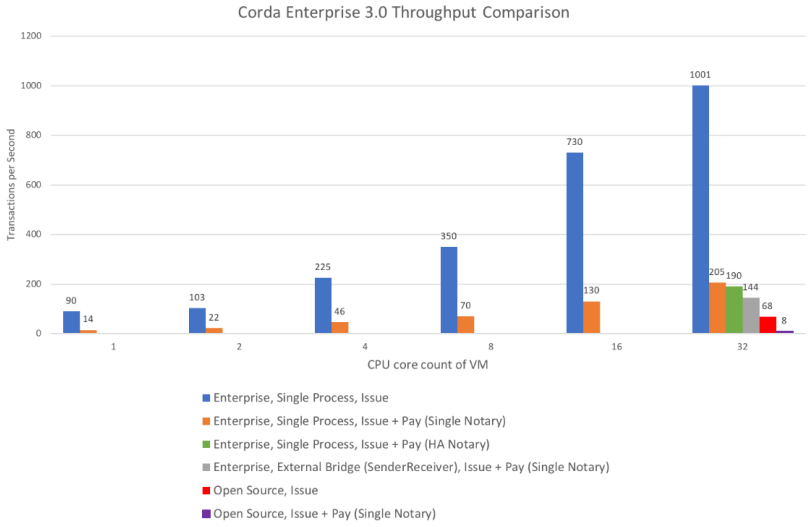

Figure 2.1: Throughput vs Number of cores

Chapter 3

Benchmarking Corda

3.1 Performance Metrics

3.1.1 Macrobenchmark Metrics

The metrics we use for benchmarking the performance of Corda are Throughput and Latency.

Throughput

Throughput in Corda is defined in terms of "business" transactions per second (TPS). Typically, this equates to a corda flow invocation and usually 1 or more corda transactions being recorded by 1 or more corda nodes, and a corda notary where required. More precisely, we define it as the quantity of a particular flow completing per second.

Different ways to represent TPS [8]:

- What a network can achieve (Network TPS)
  - Number of transactions per second in the network as a whole.
- What a node can achieve (Node TPS)
  - Number of transactions per second can a node participate.

We will be using the Node TPS as our metric as throughput. Network TPS

is not much of importance as Corda deals with point-to-point transactions and hence scales really well with the network.

Latency

Latency is defined as the duration of a transaction, i.e, for a single transaction how long does it take to span all nodes from inception to finality.

3.1.2 Microbenchmark Metrics

The metrics we use for micro benchmarking are:

- CPU Utilization
  - How much CPU is used during a transaction due to signature checking, signing etc. . .
- Memory Utilization
  - How much memory is used during a transaction depending on number of flows, number of rpc connections etc...

3.2 Benchmarking Tools

3.2.1 JMeter

JMeter is an open source java application designed to load test functional behavior and measure performance. The elements in our JMeter test plan are:

Thread Group

- Thread group elements are the beginning points of any test plan. All controllers and samplers must be under a thread group.
- The thread group element controls the number of threads JMeter will use to execute your test.

Custom Sampler for Corda RPC

By default, cordapp is shipped with a web server that is used to interact with the Corda node. The web server provides HTTP endpoints corresponding to actions in the cordapp. When you make HTTP requests to these endpoints, internally the web server makes RPC requests to the Corda node. Also the web server provided is not meant to be used instead need to be replaced with secure Spring boot server. So, we need to remove the web server and make RPC requests directly to the node since the web server takes some resources and also might add some latency.

JMeter can handle HTTP load by default but it cannot handle RPC requests that is required by the Corda. Fortunately, there is a java request interface in JMeter which we can implement to make RPC requests to the Corda node. This class is the custom sampler needed to generate load in the form of RPC requests.

3.2.2 Loggers

Corda has a solid logging & monitoring system [9]. To time a piece of code, we need to log the start and end of the code with timestamps, and later parse the log calculate the elapsed time. We can log the timestamps using two methods:

1. Writing to a file directly
    - Log parser is very simple as we only dump the info we need and in our own format.
    - This may add some latency due to blocking code but is negligible.
    - This can be used initially as PoC and later we can shift to standard loggers.
    - This is what we are using currently to time the code.
2. Standard Logging Frameworks
    - In Corda, SLF4J (Simple Logging Facade for Java) is used which provides an abstraction over various concrete logging frameworks (several of which are used within other Corda dependent 3rd party libraries).
    - Corda itself uses the Apache Log4j 2 framework for logging output.
    - Writing log parser is little complex as unneeded info is also dumped in the log files.
    - Standard logging frameworks won't add any latency as they use separate thread for writing to log file and use buffers.

- This method is to be used once we can benchmark using method 1.

We will be using method 1 in our benchmark.

3.2.3 Java Profilers & JMX Metrics

One main advantage of using JVM is the JVM ecosystem which comes with many standard profiling tools. Java profilers can be used to monitor CPU utilization, memory utilization, and JMX metrics that are exposed by the JVM process running in the node. There are many standard Java profilers in which we will be using VisualVM profiler which gives a simple UI for monitoring these metrics.

VisualVM

Java VisualVM is a tool that provides a visual interface for viewing detailed information about Java applications while they are running on a Java Virtual Machine (JVM), and for troubleshooting and profiling these applications. Java VisualVM can allow developers to generate and analyse heap dumps, track down memory leaks, browse the platform's MBeans and perform operations on those MBeans, perform and monitor garbage collection, and perform lightweight memory and CPU profiling.

VisualVM can profile remotely which is useful in our case. For profiling remotely, one needs to run jstatd on the remote host and then you can connect to that JVM process. You may also need to configure the remote system to expose JMX metrics remotely which is discussed in [10].

JMX Metrics

Java Management Extensions (JMX) is a specification for monitoring and managing Java applications. It enables a generic management system to monitor your application; raise notifications when the application needs attention; and change the state of your application to remedy problems.

JMX Monitoring is done by querying data from "Managed Beans" (MBeans) that are exposed via a JVM port. An MBean represents a resource running inside a JVM and provides data on the configuration and usage of that resource.

Corda exposes some JMX metrics (which may be increased later on) using the dropwizard.io JmxReporter facility. Corda also uses the Jolokia framework to make these accessible over an HTTP endpoint.

Prometheus

Prometheus is a systems and service monitoring system backed by time series database. It collects metrics from configured targets at given intervals, evaluates rule expressions, displays the results, and can trigger alerts if some condition is observed to be true.

One can also use prometheus to monitor CPU, memory using node_exporter [11] in prometheus and to monitor JMX metrics using prometheus_jmx_exporter [12].

3.2.4 Linux "top" tool

The top program provides a dynamic real-time view of a running system. It can display system summary information as well as a list of processes or threads currently being managed by the Linux kernel. We can use this program to monitor CPU and memory usage.

3.3 The Experiment

3.3.1 Experimental Setup

Our experiment consists of 3-party system: PartyA, PartyC, a single Notary. Both PartyA and PartyC run a cordapp and the Notary is responsible for checking the validity of transactions. Each party is run on a individual VM. Each VM has the following specs:

Specs
Intel(R) Xeon(R) CPU E5-2699C
2 cores, 1 thread per core @ 2.2Ghz
8GB RAM

Each Corda JVM process is run with 4GB heap memory for providing sufficient memory for the benchmark. JMeter runs on separate VM and is used for generating load. The whole setup is shown in figure 3.1.

3.3.2 Application Design

The cordapp used in the benchmarking is the example cordapp used in the Corda docs. This app is used to model IOUs on the blockchain. Initiator can

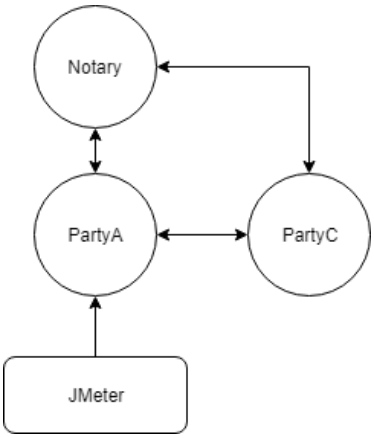

Figure 3.1: The experimental setup

provide IOUs to the acceptor of any amount. Acceptor accepts an IOU if the value of IOU is less than 100. An IOU state looks like figure 3.2.

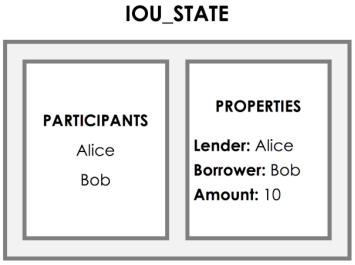

Figure 3.2: IOU State representation

IOUFlow flow will completely automate the process of issuing a new IOU onto a ledger. After a few transactions, the ledger may look like figure 3.3.

3.3.3 Experiment Design

JMeter is used to drive traffic at a node (PartyA) in the form of flow start requests. Each flow issues an IOU from PartyA to PartyC with a value of 1. To find number of transactions are completed per second, assuming all the requests from JMeter reach the server at the same time, the measurements are taken from the time when the RPC request reaches the node to the completion of the flow at the node which indicates the transaction is committed. Since all the measurements are taken in the corda node itself (not at the JMeter), the latency of the requests shouldn't affect the throughput.

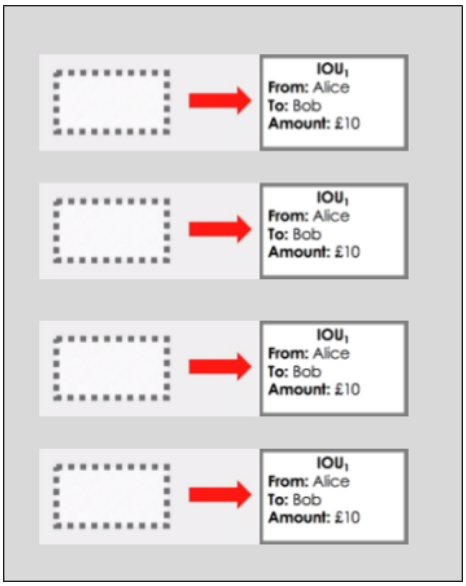

Figure 3.3: Shared ledger after few transactions

For macrobenchmark, number of concurrent users is varied from 1 to 100 using JMeter, each making a single flow request in 1 second, hence the load is varied effectively from 1 request per sec to 100 requests per sec. Throughput and latency are captured using this varying load and trends are observed. Since these metrics vary greatly on each run, each data point is averaged on 100 runs of the experiment.

For microbenchmark, 100 concurrent users each firing 1 request in a second is the load. For this load, we analyze CPU utilization, memory utilization and even parallelization in Corda.

3.4 Results

3.4.1 Macrobenchmark

Throughput

Throughput (TPS) varies with the JMeter load as shown in the figure 3.4. Throughput tends to increase as the load increases and seems to saturate around 8 tps at medium loads. Higher load than 80 requests per second is not tested. Highest TPS achieved is 8.62 tps at a load of 60 req per sec.

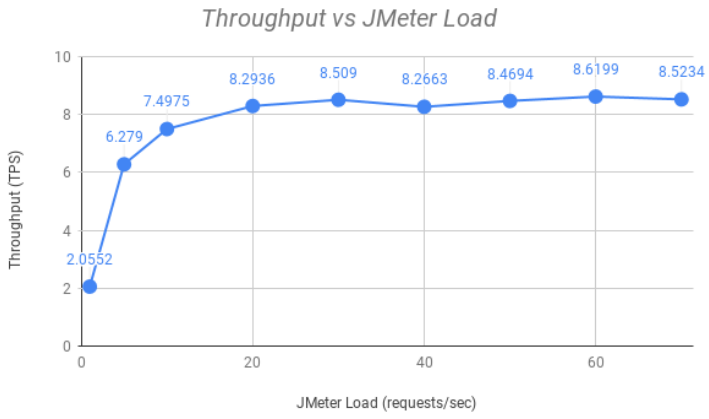

Figure 3.4: Macrobenchmark: Througput metric

Latency

Latency varies with the number of concurrent users as shown in the figure 3.5. Latency also increases as number of concurrent users increases but in a interesting linear fashion.

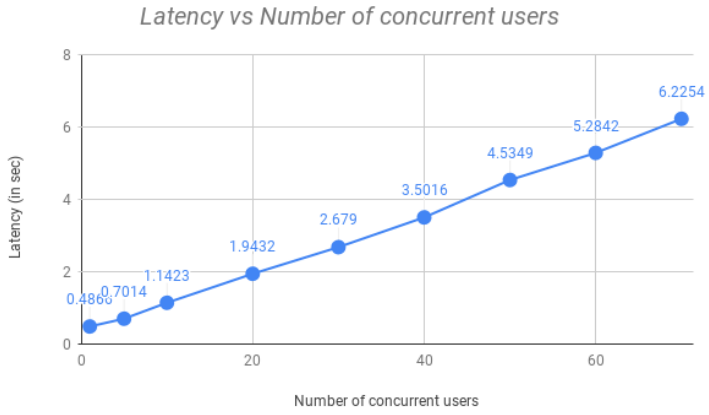

Figure 3.5: Macrobenchmark: Latency metric

3.4.2 Microbenchmark

CPU Usage

1 Iteration CPU usage is catured using "top" program in linux. As seen in the figure 3.6, CPU never utilized fully even only on 2 cores because open-source version of Corda has single threaded state machine and hence cannot use multiple cores fully.

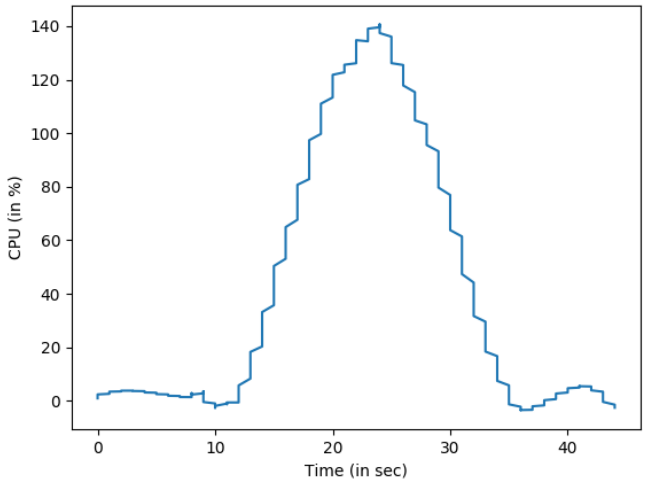

Figure 3.6: Microbenchmark: CPU usage for 1 run of experiment

4 Iterations Similar trend is seen for 4 iterations, i.e, there are 4 peaks each corresponding to a transaction as shown in the figure 3.7.

Memory Utilization

1 Iteration As seen in the figure 3.8, memory used is increased on a transaction. This is due to the additional RPC connection which takes up memory. The memory usage didn't decrease after the transaction completion because we are not disconnecting the connection after completion due to a bug in the custom sampler. This needs to be fixed.

4 Iterations For the same reason mentioned above, the memory is never decreasing even after completion of 4 transactions as shown in the figure 3.9.

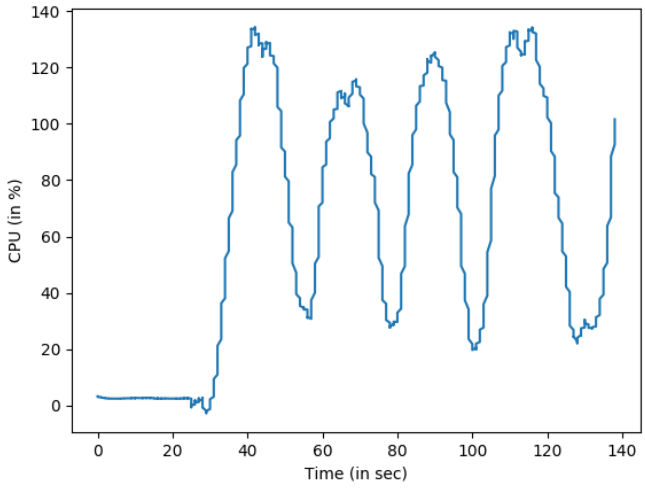

Figure 3.7: Microbenchmark: CPU usage for 4 runs of experiment

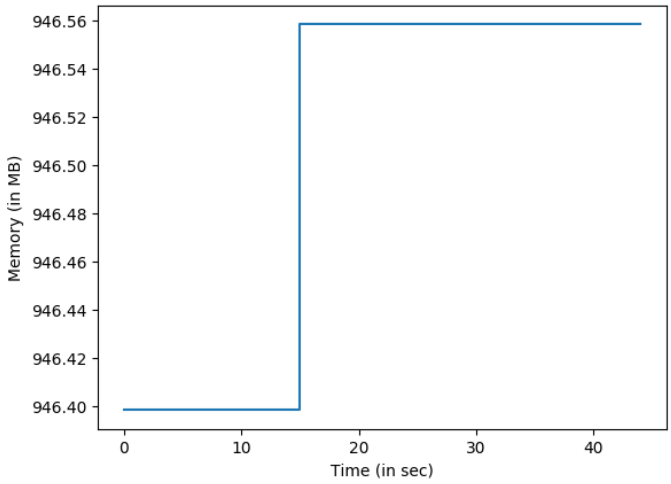

Figure 3.8: Microbenchmark: Memory utilization for 1 run of experiment

When the JVM runs out of memory due to very high number of rpc connections, garbage collector cleans up older rpc connections by dropping them.

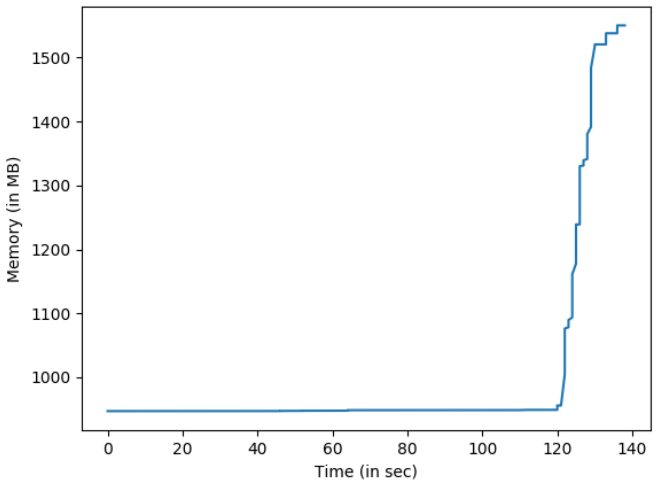

Figure 3.9: Microbenchmark: Memory utilization for 4 runs of experiment

Concurrency

Total time for the experiment: 14.3376 sec Latency: 10.7232 sec, TPS: 6.9747 tps for 100 requests in 1 second. This suggests parallelism because latency * num_requests(100) is clearly very higher than the total time. Even when the open source Corda only has single threaded state machine, parallelism is possible due to many **decoupled** stages in a Corda flow as shown in the figure 3.10. Another possible reason is because there are 4 threads handling the rpc deserialization.

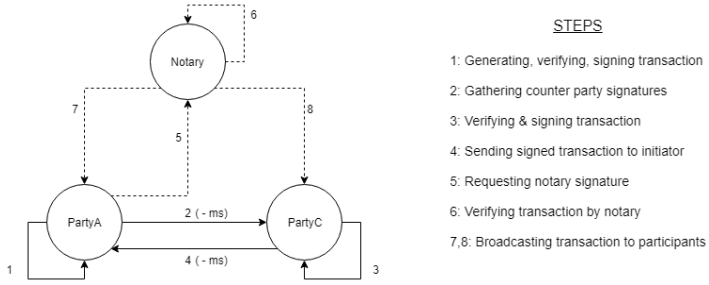

Figure 3.10: Decoupled stages in the lifecycle of a flow

Chapter 4

Future Work

In this project, we have provided many resources on how to benchmark Corda blockchain framework and are able to capture the throughput using a very simple cordapp. But this cordapp is very simple and may not resemble a real-world example. Also there are other possible metrics for capturing the performance of Corda. Here we present some of the possible improvements for a more detailed performance benchmark:

1. Using a different cordapp in the benchmarking: Finance cash Cordapp. It is a model of a cash payment. Transfer an amount owned by Node A, governed by a Cash contract, from Node A to Node B, notarized by a Notary cluster [6].

2. Our cordapp doesn't chain states in the ledger, i.e, for every new flow a new state is created in the ledger and the newly created states are never used in a future transaction. But real-world examples generally involves consuming previous states and generating new states. Benchmark using a cordapp that models these type of states.

3. Our cordapp only involves only two nodes in a single transaction. Use a cordapp whose single transaction involves more than two nodes which resembles real world transactions, for eg, bonds and cash exchange.

4. In our experiment, we assume all the requests from the JMeter reach the node at the same time which may not be true especially for high load (requests per sec). To fix this, use JMeter master-slave architecture for generating more concurrent requests per second.

5. We explored only few metrics like throughput (tps), latency, cpu usage, memory usage. Some other possible metrics are:

- Consensus Latency: Difference between the times when notary confirms for the uniqueness and other peers commit the update. Capturing this metric is very simple using the tips mentioned in section 1.x.
- Network, Disc r/w: These metrics can be captured using a standard JMX exporter like prometheus_jmx_exporter [12].
- ArtemisMQ Queue Length: This metric is exposed via JMX in Corda. So, monitoring this metric is simple but it still gives a very good insight on how requests are queued especially on server side.

6. Do microbenchmark using a standard java profiler like VisualVM for a better output.
7. Logging is currently done using Java File API which may add some latency due to blocking call. To circumvent this, switch to logging via the standard logger in Corda and write a new log parser accordingly.
8. Current benchmark is done on Corda version 3.3 and many improvements are made in the current Corda version 4.0. To benchmark the latest version, though the most of the code remains same, we need to update the cordapp to make it Corda 4.0 compatible and you may also need to update instrumentation depending on the version changes.

Also note that there is a bug in our JMeter custom sampler that causes error if we close the rpc connection after the flow is completed. This is needs to be fixed. This is a probable reason why memory usage always increases as mentioned in section 3.4.2.

Bibliography

[1] "Messaging in corda." [Online]. Available: docs.corda.net/messaging.html

[2] "Corda rpc protocol." [Online]. Available: github.com/Nihhaar/corda/blob/benchmark/node-api/src/main/kotlin/net/corda/nodeapi/RPCApi.kt

[3] "Node services in corda." [Online]. Available: docs.corda.net/node-services.html

[4] "Corda node implementation." [Online]. Available: github.com/corda/corda/blob/master/node/src/main/kotlin/net/corda/node/internal/Node.kt

[5] "Corda client rpc." [Online]. Available: docs.corda.net/releases/release-V3.3/clientrpc.html

[6] R. Parker, "Enterprise corda: Journey to 1000 tps." [Online]. Available: www.slideshare.net/MarketingTeamr3/devday-corda-enterprise-journey-to-1000-tps-per-node-rick-parker

[7] "Sizing & performance in corda." [Online]. Available: docs.corda.r3.com/sizing-and-performance.html

[8] M. Ward, "Transactions per second (tps)." [Online]. Available: medium.com/corda/transactions-per-second-tps-de3fb55d60e3

[9] "Logging & monitoring in corda." [Online]. Available: docs.corda.net/design/monitoring-management/design.html

[10] "Visualvm jmx connections." [Online]. Available: docs.oracle.com/javase/7/docs/technotes/guides/visualvm/jmx_connections.html

[11] "Prometheus node exporter." [Online]. Available: github.com/prometheus/node_exporter

[12] "Prometheus jmx exporter." [Online]. Available: github.com/prometheus/jmx_exporter

Appendix A

Building Corda

Listing A.1: Building Corda

```
mkdir ~/workspace && cd ~/workspace
git clone https://github.com/Nihhaar/corda.git
git clone https://github.com/Nihhaar/cordapp-example.git
wget https://archive.apache.org/dist/jmeter/binaries/apache-jmeter-5.0.zip
git clone https://github.com/Nihhaar/jmeter-rpc.git
git clone https://github.com/Nihhaar/Corda-Blockchain-Benchmark.git

# Build corda and install it to the local maven repo at ~/.m2
cd corda
./gradlew install
cd ..

# Build cordapp along with corda to deploy nodes
# Configure IP addresses in cordapp-example/kotlin-source/build.gradle
# Nodes are created in cordapp-example/kotlin-source/build/nodes/
cd cordapp-example
./gradlew deployNodes
cd ..
```

Appendix B

Running Corda Nodes

Listing B.1: Running Corda Nodes

```
export TERMINAL=gnome-terminal
cd cordapp-example/kotlin-source/build/nodes/
# Start the nodes
cd Notary
$TERMINAL -e sh -c 'java -jar corda.jar;zsh' &> /dev/null &
cd ..
cd PartyA
$TERMINAL -e sh -c 'java -jar corda.jar;zsh' &> /dev/null &
cd ..
cd PartyB
$TERMINAL -e sh -c 'java -jar corda.jar;zsh' &> /dev/null &
cd ..
cd PartyC
$TERMINAL -e sh -c 'java -jar corda.jar;zsh' &> /dev/null &
cd ..
cd ../../../../
```

Appendix C

Setting up JMeter

Listing C.1: Running JMeter with custom sampler

```
# Build jmeter sampler
# Output can be found at build/libs/jmeter-rpc-1.0-SNAPSHOT.jar
cd jmeter-rpc
./gradlew build
cd ..

# Copy runtime libraries required for JMeter sampler
mkdir libs
cd libs
cd ~/.m2/repository/net/corda/corda/3.3-corda-local
cp corda-3.3-corda-local.jar ~/workspace/libs
cd ~/workspace/libs
unzip corda-3.3-corda-local.jar
rm corda-3.3-corda-local.jar
rm groovy-all-1.8.9.jar
cd ..

# Unzip JMeter
unzip apache-jmeter-5.0.zip -d apache-jmeter-5.0
vim apache-jmeter-5.0/bin/user.properties
# Edit user.classpath variable to libs directory. For eg,
# user.classpath=/home/nihhaar/workspace/libs/
```