COMP 512 Distributed Reservation System overview.

Project Report III

Submitted by

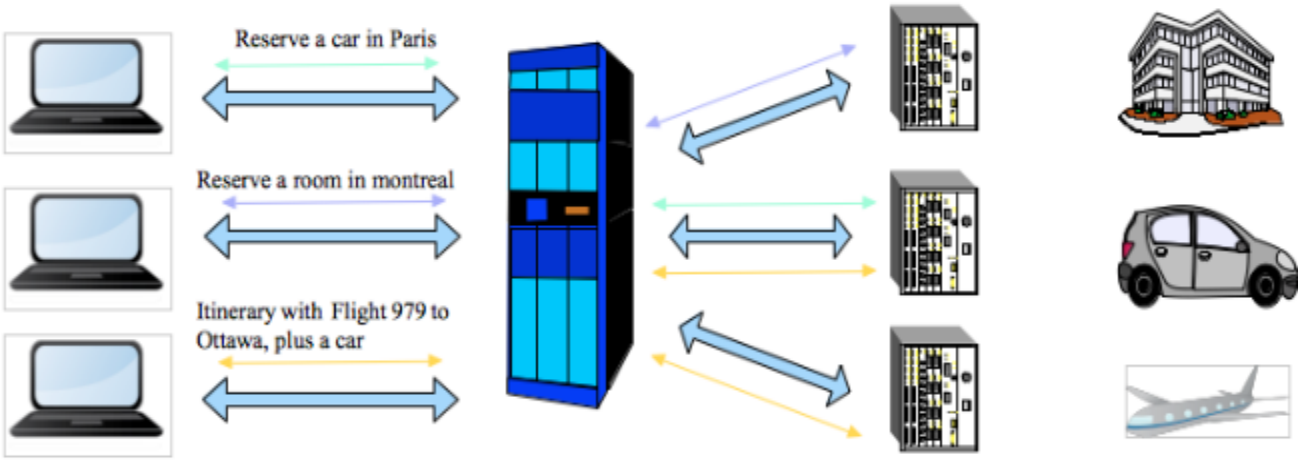
Joseph Vinish D'silva
Dhirendra Singh

Project Check Points List

Project I - Basic TCP / RMI - 3 Tier Architecture.

Three tier architecture of the reservation system.

The High level diagram of the 3 tier architecture of the reservation system is as depicted in Figure below.



Here the clients connect to a well known middleware layer, and is oblivious to the existence of multiple resource managers that actually implements and executes the various requests. Clients pass their requests to the middleware layer who executes the requests on the various reservation systems on behalf of the client.

The middleware layer in turn is aware of the various resource manager servers that implement different functionality. It knows that it needs to direct the hotel reservation requests to the hotel resource manager, car reservation request to car resource manager and so forth. It also takes care of any requests that spans across the three reservation systems. Hence its capable of looking at an itinerary request and splitting it down to flight reservation, car reservation and hotel room booking. The middleware layer also provides a consolidated view of the customer info, in terms of the person's billing by collecting the reservation information of a particular customer from all the three reservation systems.

The customer information is duplicated across all the three RMs as all of them need access to customer information to process reservation requests.

Java RMI Implementation

In the Java RMI version of the implementation the client systems are aware only of the Middleware layer interface which provides an abstract view of all the reservation functionalities available in the system. Clients obtain a remote reference to the middleware layer object which has implemented the functionalities and manage their reservation requests through that object.

The middleware layer object knows about the three resource management servers and have access to interfaces that abstract their functionality. The middleware layer thus obtains remote references to these three objects and implements its own advertised middleware layer functionality by redirecting the requests to these three objects as needed and returning the output results to the client process.

The data of the system resides completely on the three resource management servers. The reservation system support concurrent execution, by virtue of java RMI system, and as such the data integrity mechanism is maintained by the synchronization of the access to data items in the resource manager servers. This is accomplished by making use of the synchronization support of Java, which enables thread safe components to be developed.

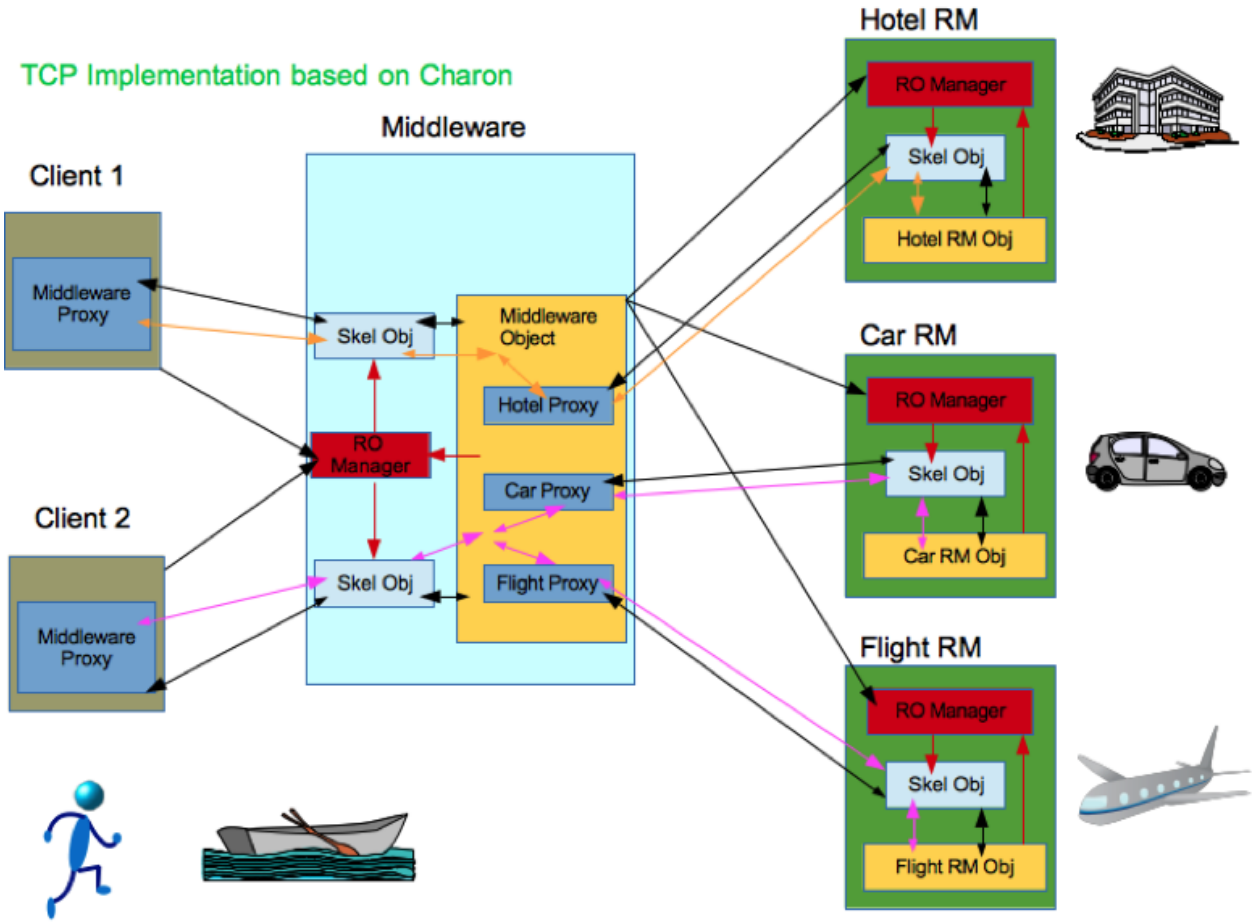
TCP Implementation

The TCP version of the reservation system was created by using a custom developed tool called charon. How charon works and the development methodology of using charon will be explained in the sections later.

In the TCP version, the remote object is implemented via an interface similar to the Java RMI implementation. However there are proxy classes which also implement the object interface to which the client has access to. These proxy classes are capable of opening network connections to the server side which has the real object implementation. The server process is also equipped with skeleton classes that services any request coming from the client side.

As in the case of Java RMI implementation, the client knows and have access only to the middleware layer via a middleware proxy object. The proxy object is in charge of communicating any client requests with the skeleton object residing on the middleware server. The skeleton object has access to the actual middleware implementation which then executes the requests on various reservation systems as is needed. In order to facilitate this remote execution, it makes use of the same proxy – skeleton architecture to interact with the resource manager objects residing in remote hosts.

Similar to the Java RMI implementation, charon's remote object management system is capable of supporting concurrent requests making it possible for multiple clients to connect and issue requests at the same time. Thus data integrity is ensured by using synchronization at the resource manager servers.



Design principles of Charon

The purpose of developing Charon was to implement an automated mechanism that, given an interface definition can provide a complete system that will generate not only a proxy class and a skeleton class which the client program and server program can correspondingly

utilize, but also to supply a complete runtime support to loading those definitions and managing the objects transparently without having the programmer to explicitly write the functionality to do so. By achieving this we were able to reduce the effort to build a TCP based software to that of using corresponding Java RMI functionality.

In order to achieve complete automation, we built up on the powerful support that Java provides in reflection package both in terms of static interpretation of compiled class definitions as well as runtime loading of classes and object instantiations.

The starting point of Charon is a definition of an interface, which we are intending to develop for remote access. This interface definition is next compiled, after which we will execute Charon to build the proxy and skeleton classes for us.

The programmer is not required to edit the Proxy and Skeleton classes. In the server program, the programmer creates the actual object that implements the interface definition and then registers it with Charon's runtime Object Manager. This Manager is responsible for receiving any requests to interact with the object.

The Client program invokes Charon's runtime Manager static methods to get a Proxy reference to the server Object. These methods can interact with the Object Manager in the server and identify the name of the proxy class to be loaded etc. The client program is then returned a reference to this proxy class. The client program can thus execute the function as though it was locally available (save for those remote exceptions). The programmer effort is thus comparable to RMI implementation.

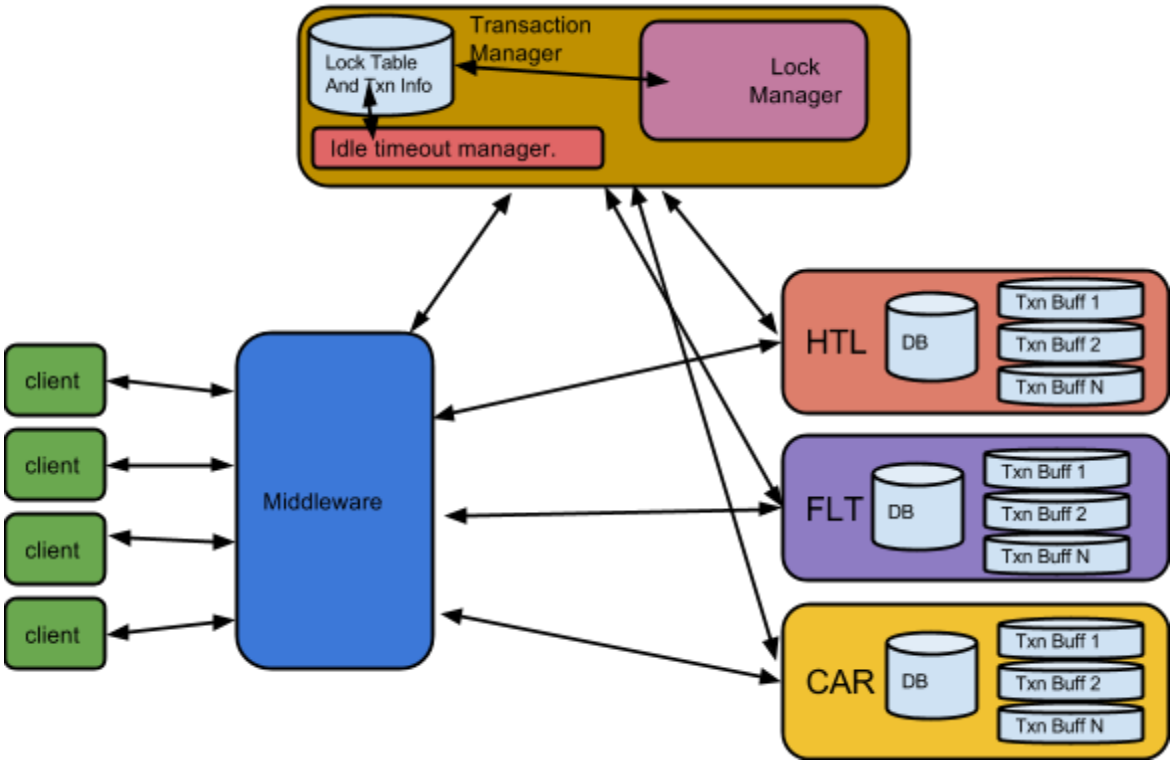
The details and example programs of how to use Charon is available in project report - I.

Project II - Transaction Management in DS.

System Architecture

In this project, we implement the basic transaction semantics into the RM distributed systems. The effort provides consistency of data and operations, on a non-failure scenario of RMs and Middleware.

The high level system architecture of the reservation system is as depicted in the diagram below



Client request are sent to the middleware that communicates with the Transaction Manager and the resource RMs.

The centralized lock manager forms an internal component of the Transaction manager.

When a client initiates a transaction, the middleware forwards the request to the transaction manager that generates a unique transaction id and returns it to the client. Any further communications from client references this transaction id.

When an RM receives a request, it will look in its transaction buffers whether the transaction is present. If not, it will ask the Transaction Manager to register itself as a participant in that transaction.

The RM also issues a lock request for the object requested, to the Lock Manager and waits (till DEADLOCK timeout) for a lock. Upon obtaining a lock successfully, it will continue to process the data.

For read requests, RM first checks if the data is available in the transaction specific buffer. If found it's returned from the transaction buffer. Otherwise the data is read from the main data store and returned.

For write requests, the data is read into the transaction buffer if not already present (a new copy is made) and this copy is returned. All modifications are performed on this copy.

When the client initiates a commit, middleware passes on the request to transaction manager, who in turn sends it out to any RMs that had registered for the transaction. The RMs flush the copy from the transaction buffer for any modified data and then discards the transaction buffers.

For aborts, either initiated by the client or by the Transaction manager for idle timeouts, the RMs simply discard their transaction buffer.

The Transaction Manager also keeps track of transactions that has been idle for a while and aborts it freeing the locks. If the client at some point in time tries to continue, a transaction aborted exception will be thrown.

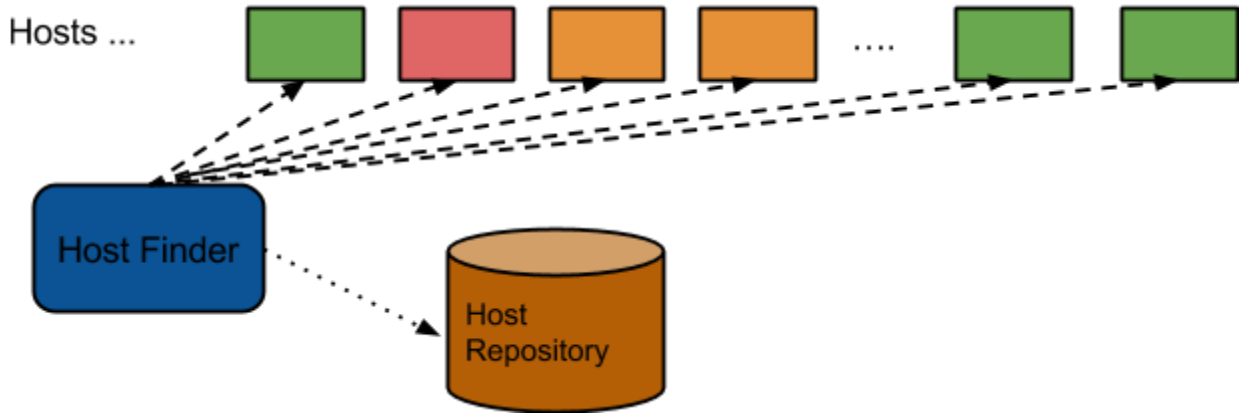
The deadlock and idle timeout intervals are externally configurable.

Reserve Itinerary.

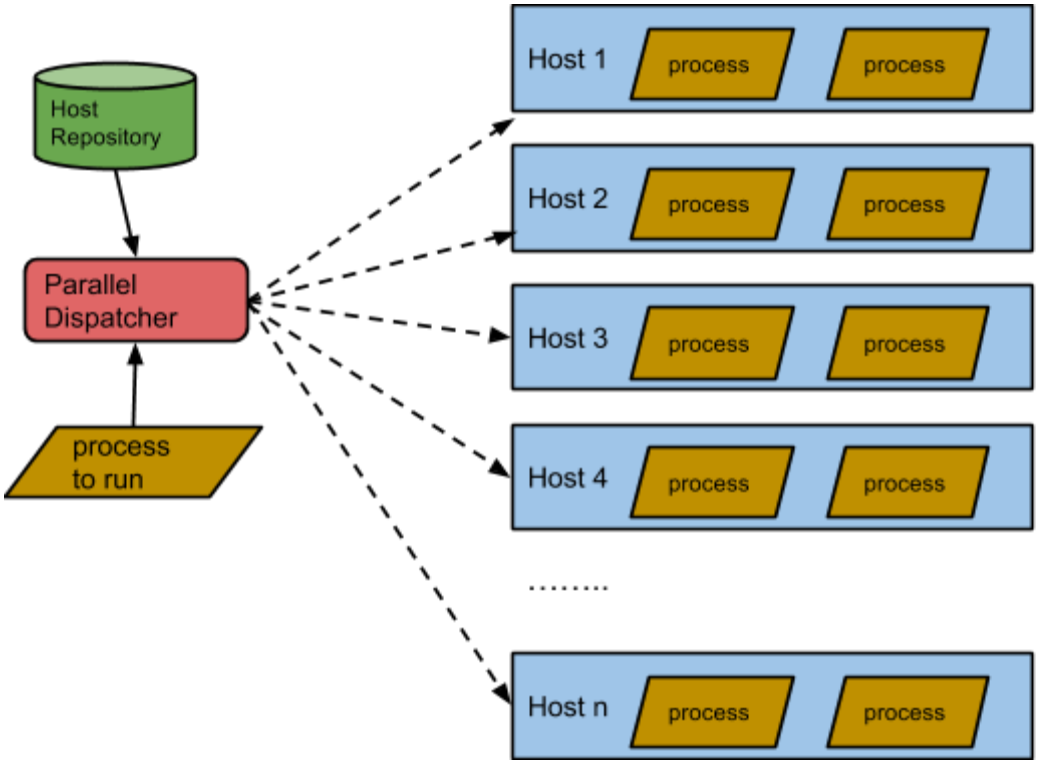
Reserve itinerary has special implementation in the sense that the middleware always request the RMs to confirm the availability (who puts a write lock), however no actual modification takes place at this step. If all the RMs confirm on availability then the changes are made. If any RM is not able to confirm the availability of resources, no modifications are made and the locks are released when the transaction is committed/aborted.

Test Environment setup.

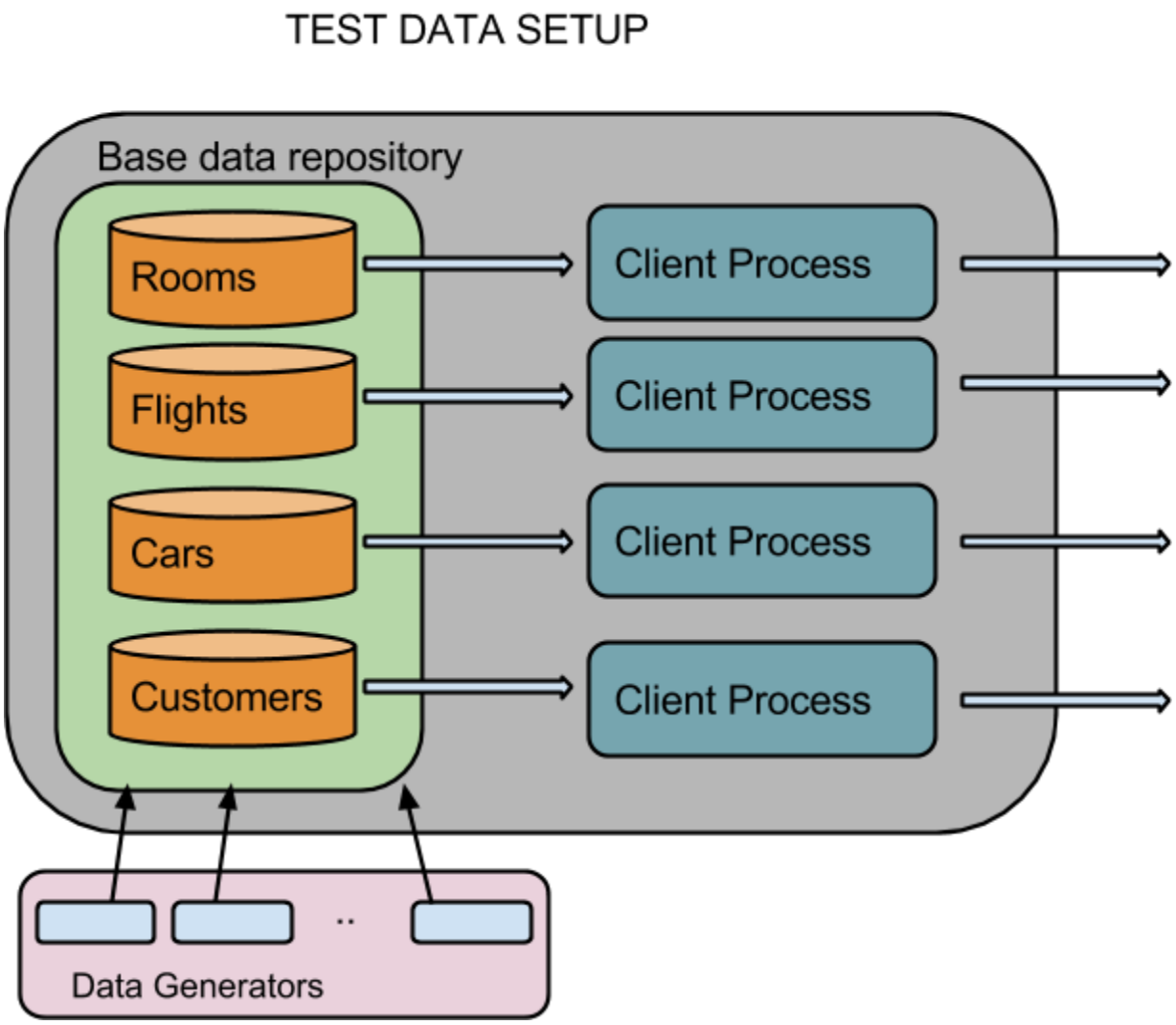
The infrastructure for parallel test setup is as described below



The host finder can be used to locate hosts in the network and their load averages to pick suitable hosts for parallel testing environment.

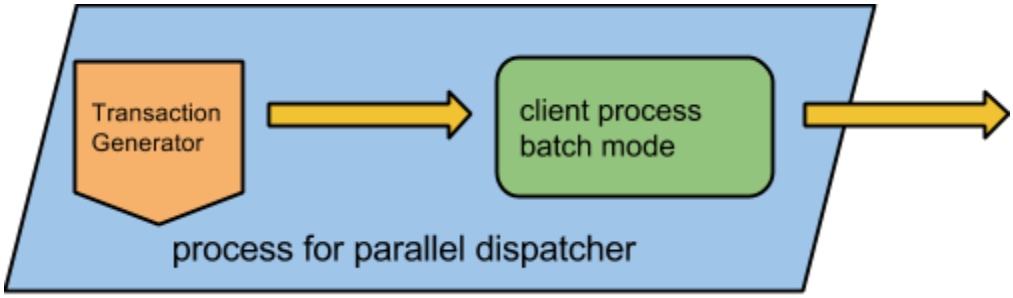


The parallel dispatcher is a program that can execute multiple instances of a script in multiple servers depending on the amount of parallelism required. We make use of this process to have multiple clients launched from different servers.



The base data repository is used to refresh the test environment to a known configuration of data. All test cases will be executed on top of this configuration. This way, we can ensure that all test cases have a known starting point of data configuration.

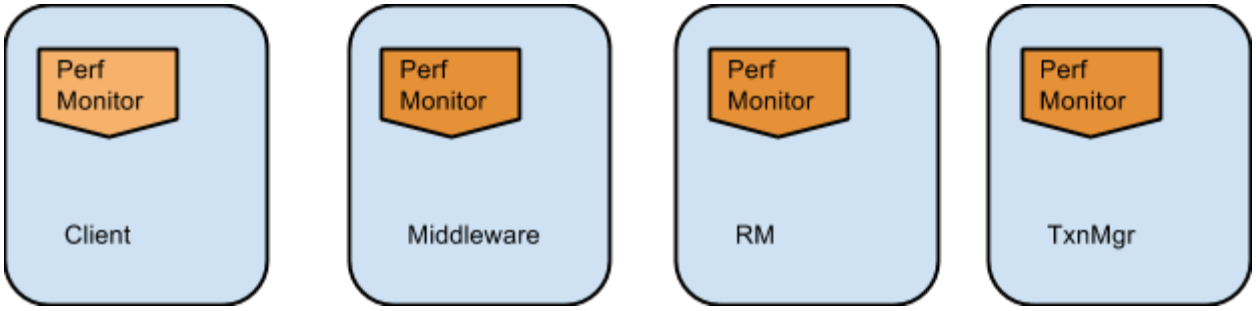
The transaction generator is used to generate transactions that are then fed to the client process, which can be configured to run on batch mode. In this mode it is possible to pass a option to limit the number of transactions that are executed per unit time.



The Performance monitor is a component embedded into all of the elements in the reservation system. It is used to record the starttime and endtime of an operation and print out the response times upon completion of the operation. This information is in CSV format. That can be used by charting utilities like Excel, MATLAB etc.

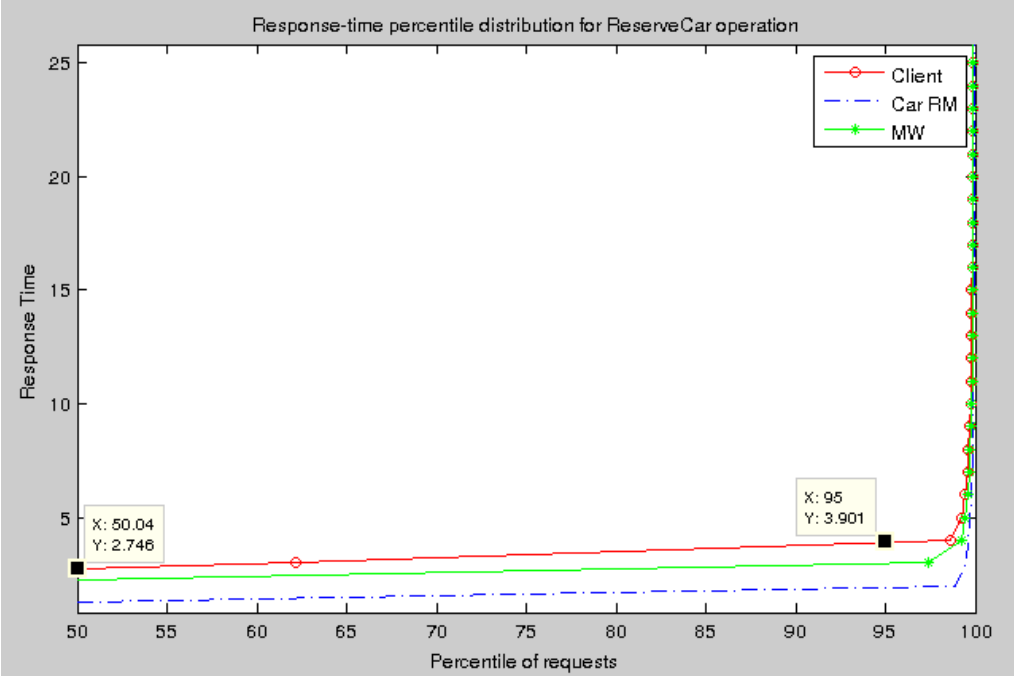
The component can be enabled/disabled externally by setting the appropriate environment variable.

The test suite can be seeded by a test identifier, in which case all the test data generated will be tagged with it. This makes it easier to identify a specific test case with each a performance record is to be identified with.

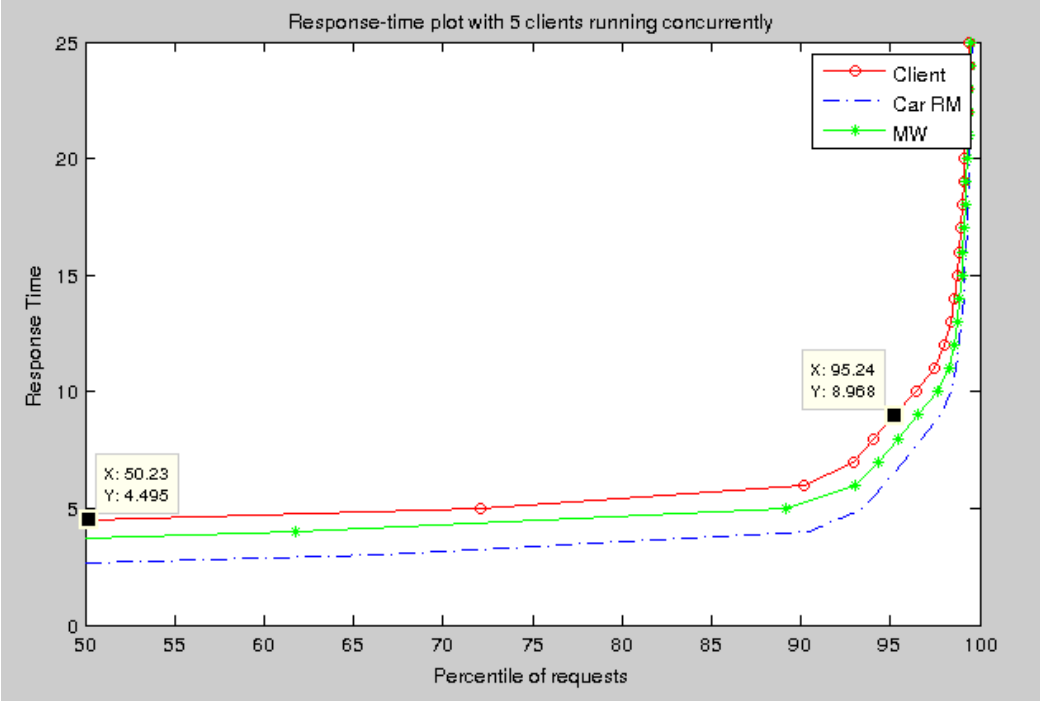


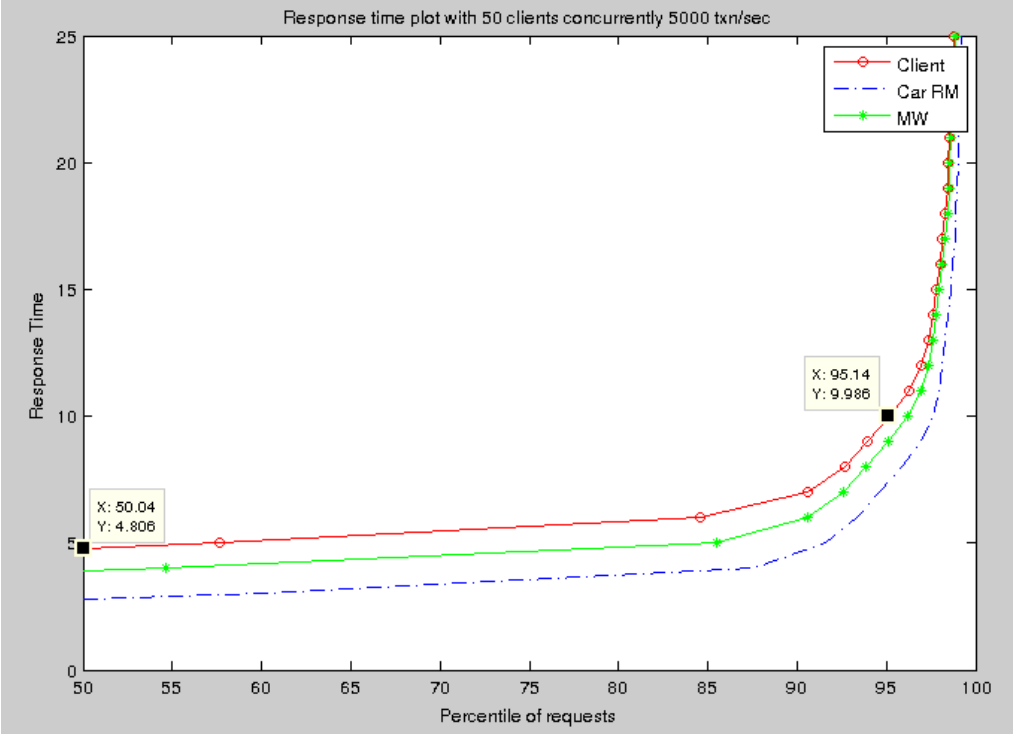
Test Execution

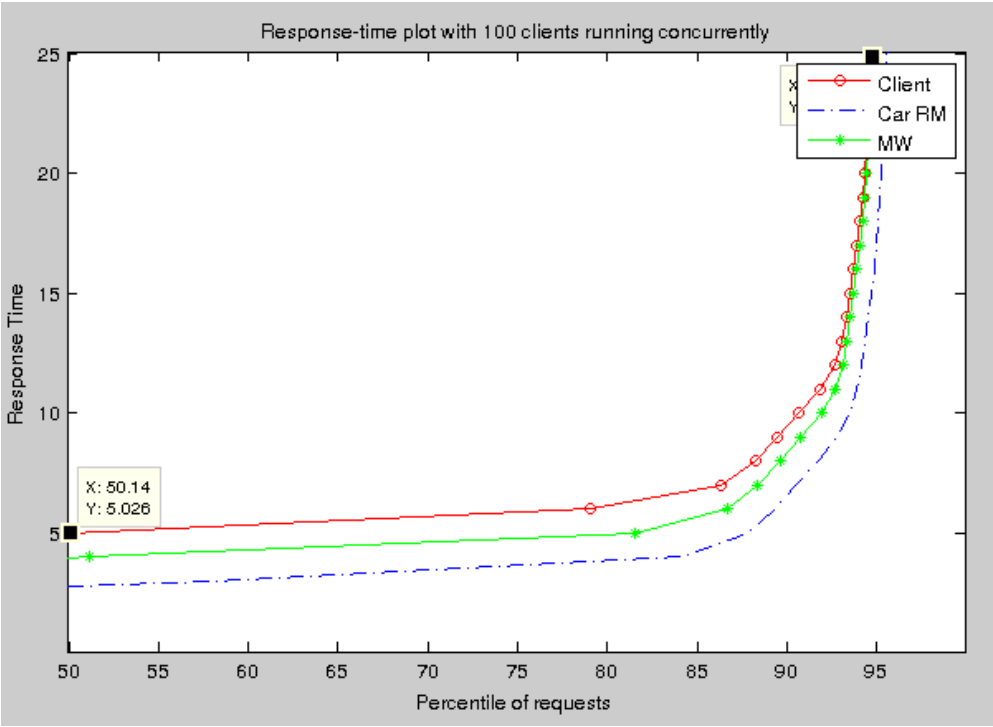
By executing a single client without any wait period between transactions, it was noticed that 50% of the requests completed under 2.75ms and 95% under 3.9ms



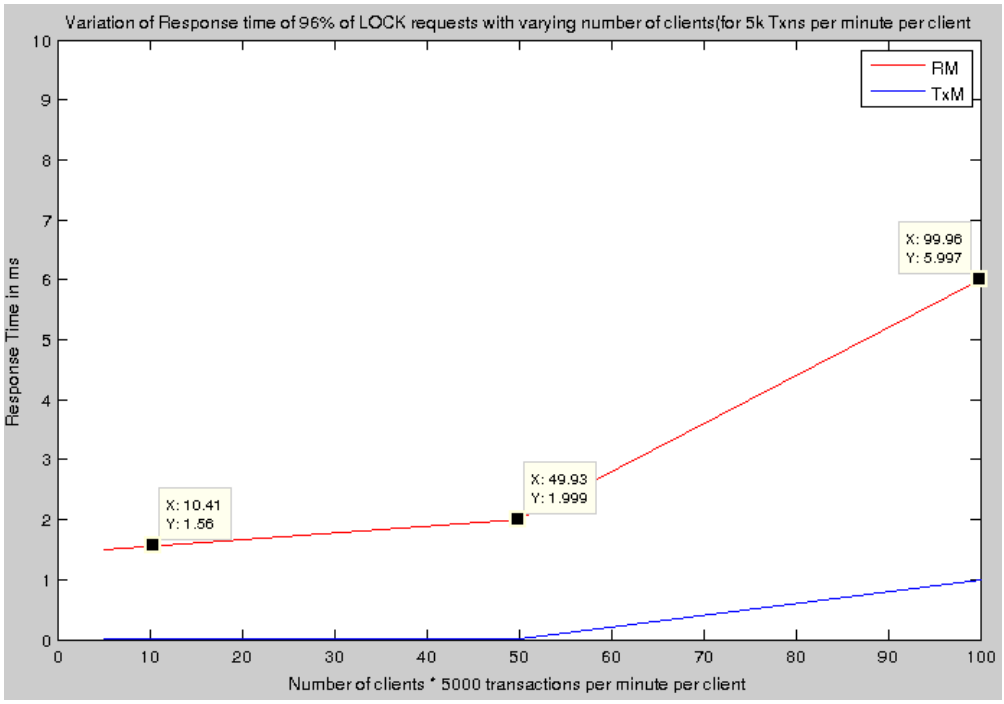
Execution of 5000 txn / Sec



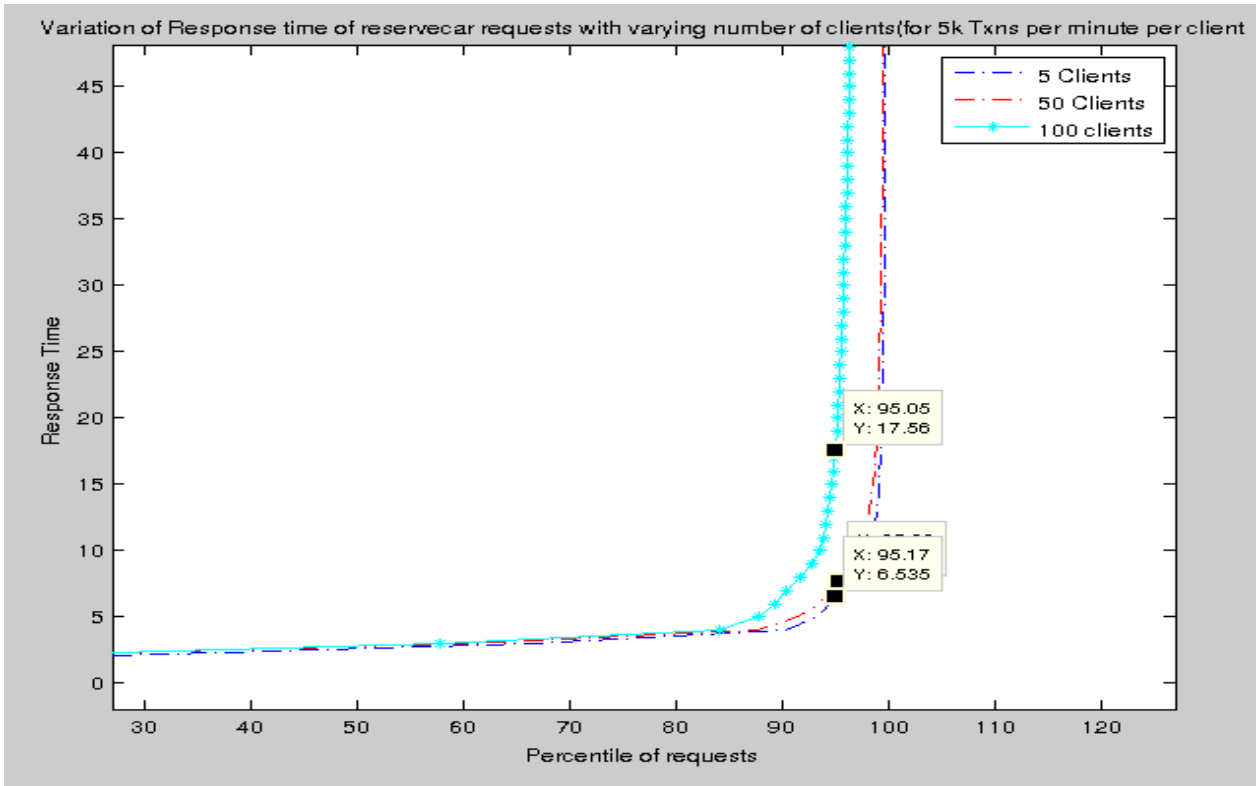




We notice as the number of clients increased from 5 to 100, that 50% of queries still give good performance, however the outliers at 95% does increase. While for 5 to 50 the changes are not significant, the effects are more noticeable as we increase the clients to 100.



We can notice the proportional increase in lock request time in the system, pointing to lock contention and deadlocks occurring in the system at higher concurrency.



Project III - Replication and Fault tolerance in DS.

Due to the complexity of the DS Architecture with replication, we will investigate the system design at various steps and then take a look at high level architecture.

Jango - Building a Replication Framework

The core of the DS replication system is Jango, a custom package that we developed which provides a lot of basic failover/fault tolerance primitives. This package consists of two run time components and a tool that can generate the stub files to be used by the client application to switch between replicated servers.

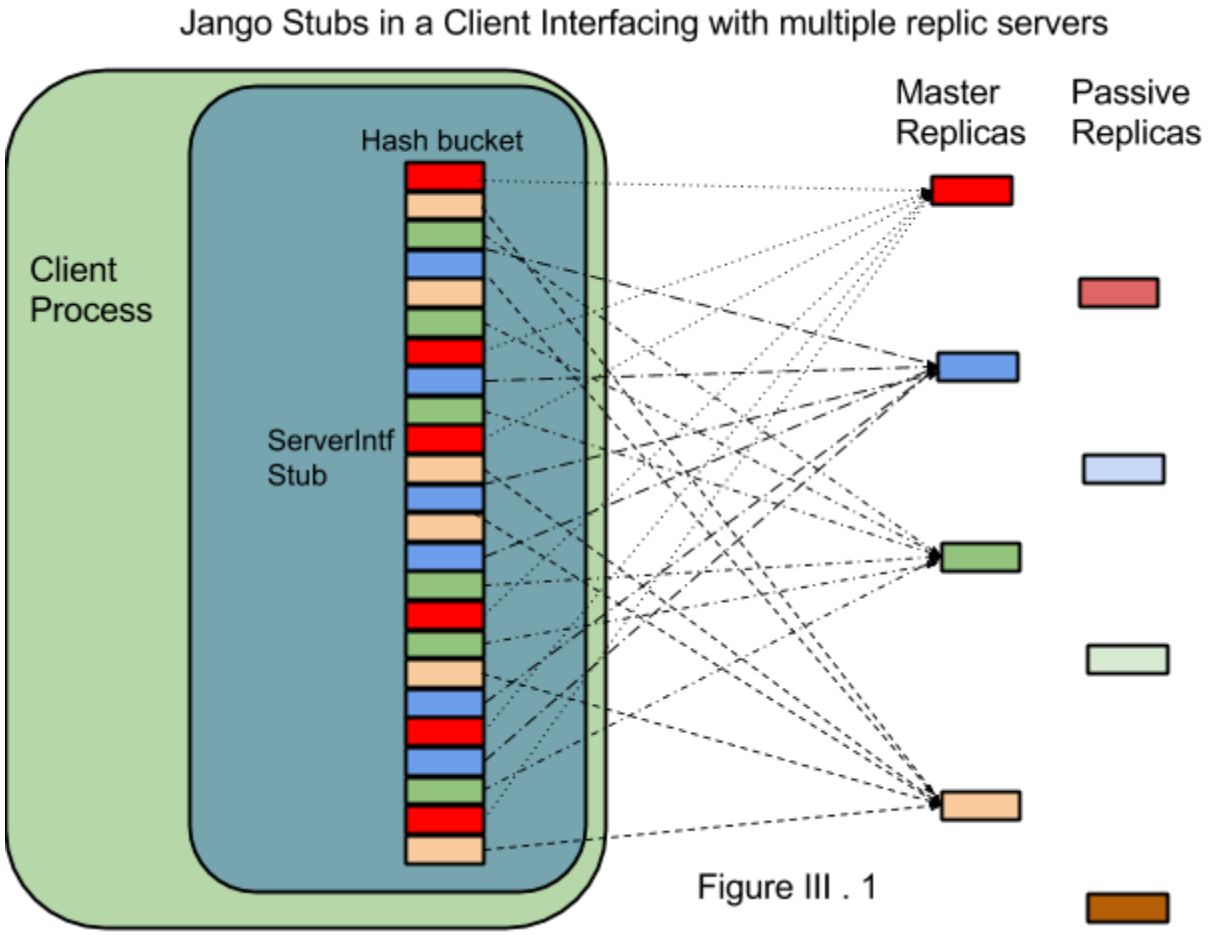
JangoRepStub - Transparent Fail over at the client side.

Jango's primitive stubs are capable of switching between replicas in a way that's transparent to the client executing a method using the stub. The stubs themselves are generated by running the tool provided by Jango over the server object's interface which is exported for remote invocation. As such the stub implements the server interface, which can then be made available to client programs. The stubs can be configured for making use of load balancing if there are multiple master servers available.

The stubs work with RMI to understand the replicated servers available for a particular service, and probes to see which of them are masters and which are acting as passives. It then builds a hashbucket based on virtual nodes and consistent hashing to load balance the requests between various available replicas. The hashing itself is performed on the arguments passed on to the methods, which can be configured while generating the stubs. Thus for example it is possible to hash on transaction ids when available, thus making sure that the requests from the same transactions are processed by the same server, if it is available throughout the lifetime of the transaction. By making use of consistent hashing, the adding or removing of replicas will result only in minimal amount of transaction migration between replicas.

The stubs can also detect failures and notify the passive servers, if there are no masters available. If a stub detects a failure for one of its active replicas in the hashbucket while executing a remote method, it will rebuild the hashtable and probe all the known servers. This also provides an opportunity to understand if any new master server has become available and add to its hashbucket. Once the hashbucket is rebuilt, it will execute the function again. The stubs assume that the function execution is idempotent and deterministic at the server side.

The figure below shows at a high level how the client uses a stub to interact with many replicas via a stub. We will be using the below configuration for our client - to - Middleware implementation as our Middleware is idempotent (being stateless). There won't be any passive replicas in our Middleware configuration though.



JangoRep - A basic replica communication primitive for the Servers.

JangoRep is basic framework that can be used to build Servers that needs to form a group. The framework does not provide any guarantees of idempotent execution or data replication, instead serves as a basic interaction mechanism between Servers providing the same functionality.

The framework thus provides mechanisms to identify and register between replicas of the same group, identifying the masters and passives, provisions to load data from a master server at startup and mechanisms to lock replica at different levels, and to send and receive notifications from other replicas. It also provides an interface to the JangoRepStubs to interact with and to verify status and receive alerts.

Applications can be built by extending this framework and providing its own data synchronization mechanisms to keep the application specific data in sync between the various replicas. The framework is designed in such a way that in case of no masters are detected by a Stub or a replica (which is booting up), it alerts the first available passive in the order of replicas (priority) to switch to master. This replica will inturn do a check on its neighbours to ensure that there are no masters and switch to master. A replica that boots up and finds no neighbours assumes itself to be a master. A replica that is booting up will connect to an available master and load data into its own repository. The master locks down itself into exclusive mode during its data transfer to the new replica, and also registers the new replica with itself for any future data transfers as part of data synchronization. The new replica then continues to register itself with other replicas that is available (passive or active) and then depending on whether the configuration is for a single master or many master will go into master / passive mode.

The figure below depicts a configuration with one master and many passives interfacing between themselves and the clients that are connecting to it with the stubs. This will be the configuration we will be using in our replication implementation of RMs and Transaction Managers.

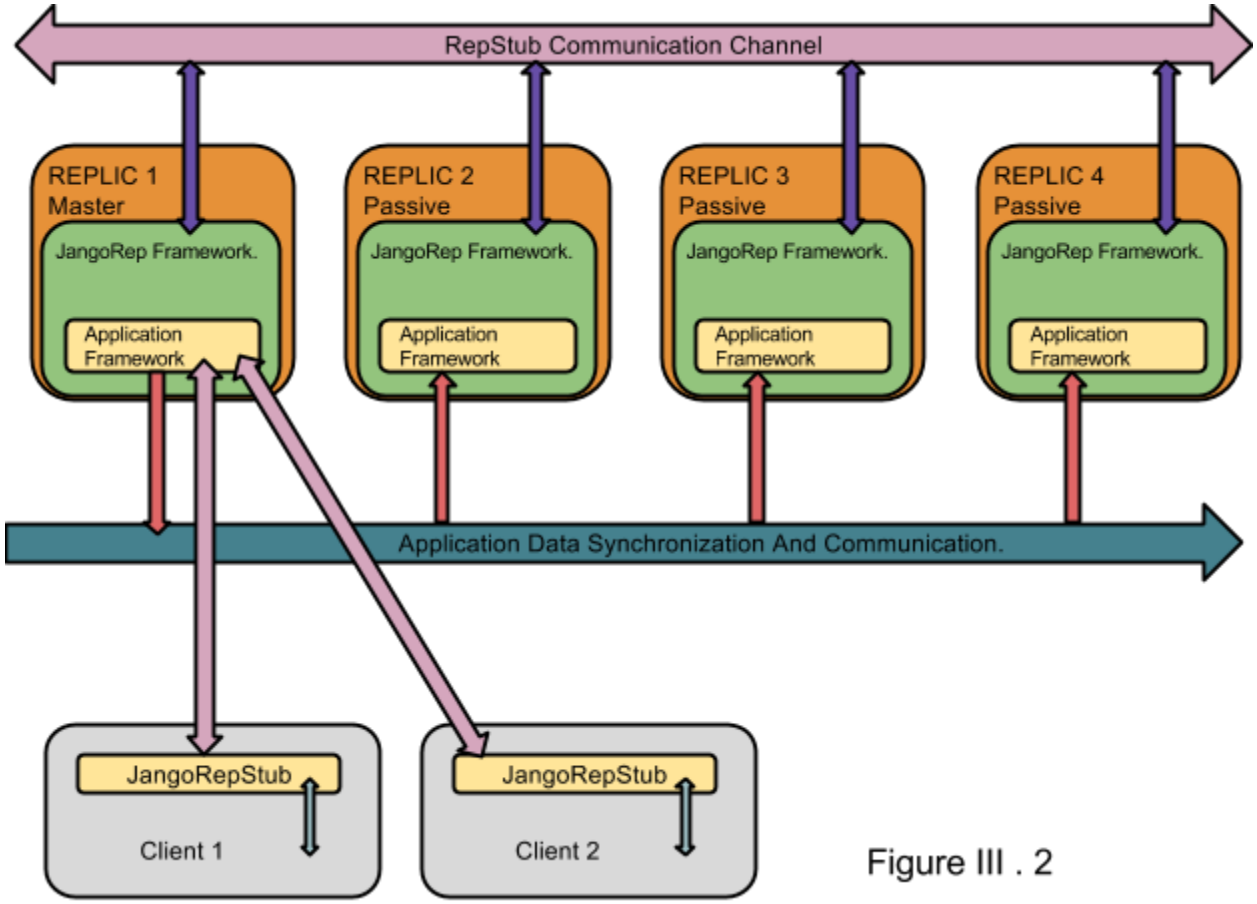


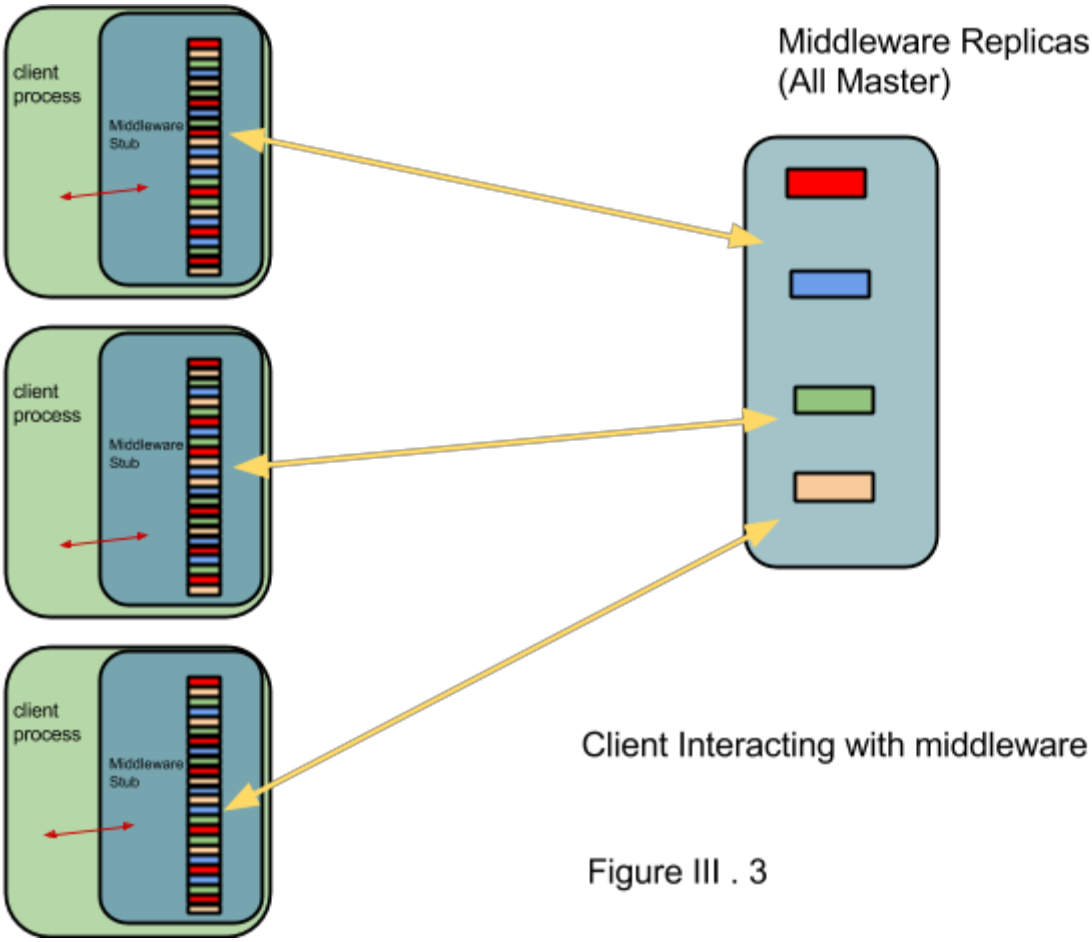
Figure III . 2

System Architecture

Now that we have discussed the core architecture of Jango Framework, we will see how various components of the Distributed Reservation system are integrated.

Client - Middleware Interaction

The client uses the middleware stub to issues it's reservation requests, these reservation requests are then redirected by the stub to one of the available middleware replicas. If a request fails due to the replica being not available, (RemoteException) the stub will rebuild its hashbucket by probing all replicas and re-issue the request. The results are passed back to the client. It should be noted that the stubs will return any exception other than a RemoteException. This is because the other exceptions are considered part of the normal workflow.



Client Interacting with middleware

Figure III . 3

Middleware - RM and Middleware - Transaction Manager Interaction

The RMs and Transaction Managers run in a single master multiple passive replica configuration. As such there's only one master at any time for a given RM type and the TM. Middleware makes uses of stubs designed for RM and TM (Generate by Jango) that can automatically detect the master in each group and connect to it. If the stub fails while invoking an operation on the master replica, it triggers a lookup on all the replicas in that group to see if there's a new master, if it still can't find one, it will invoke the alert function on the first passive replica in the order of priority and ask it to become the master. It will then rebuild the hashbucket and then execute the method on the master again.

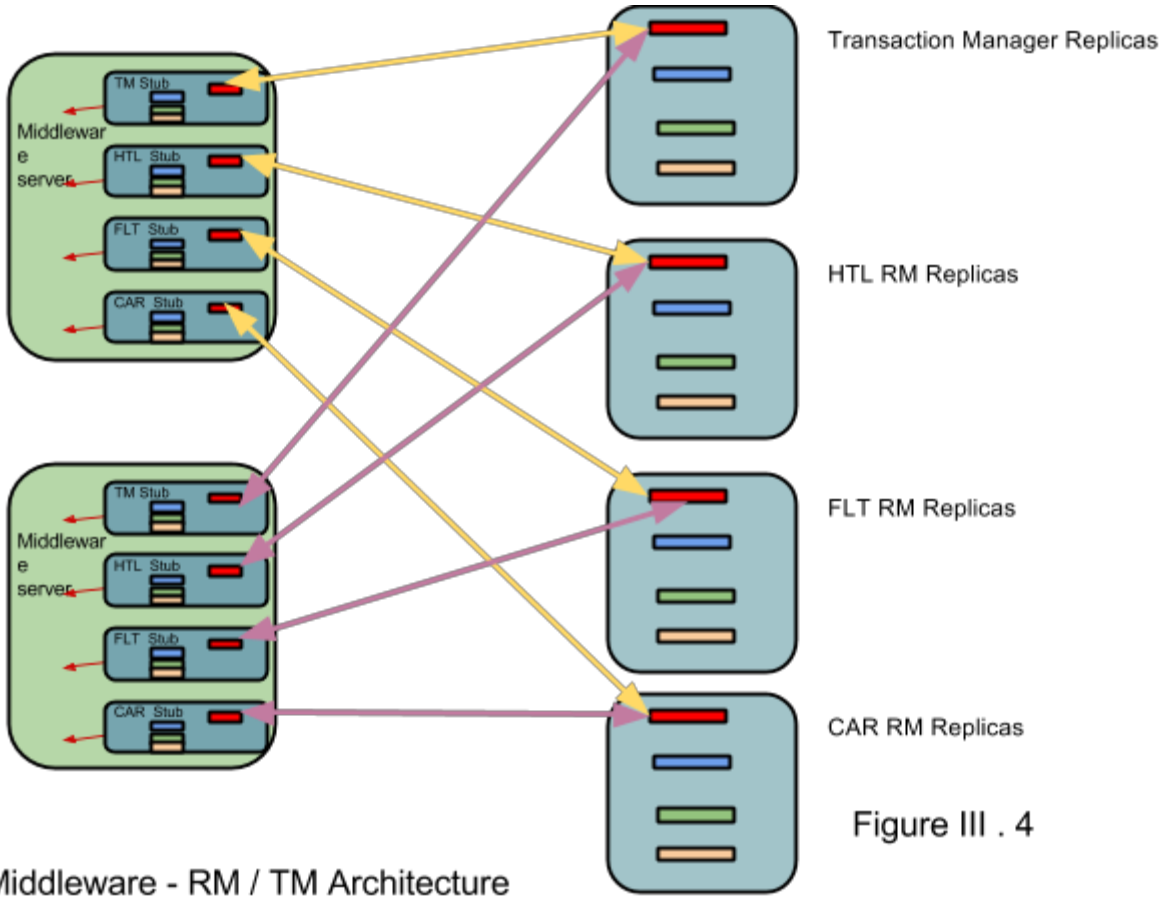


Figure III . 4

Middleware - RM / TM Architecture

RM - Transaction Manager Interaction

An RM acts as a client of TM , thus RMs will have stubs meant to interact with the TM, thus if at any point an operation on TM fails, the stubs will automatically trigger its mechanisms to contact the next passive RM for fail over.

An RM, at bootup will register with the Transaction Manager indicating what RM group its representing (ie HTL/FLT/CAR), as a consequence any abort / commit involving that RM group will be transmitted by the TM to that RM. (TM doesn't bother if an RM goes down, it's only responsible for pushing the abort/commit alerts to all available RMs in that group).

When the master RM for an RM group receives a request, if it finds that this is the first time that RM type is involved in that transaction, it will ask the TM to register this RM group for that particular transaction id. The TM, will then send any commit/abort messages to all the RMs registered for that group as and when required.

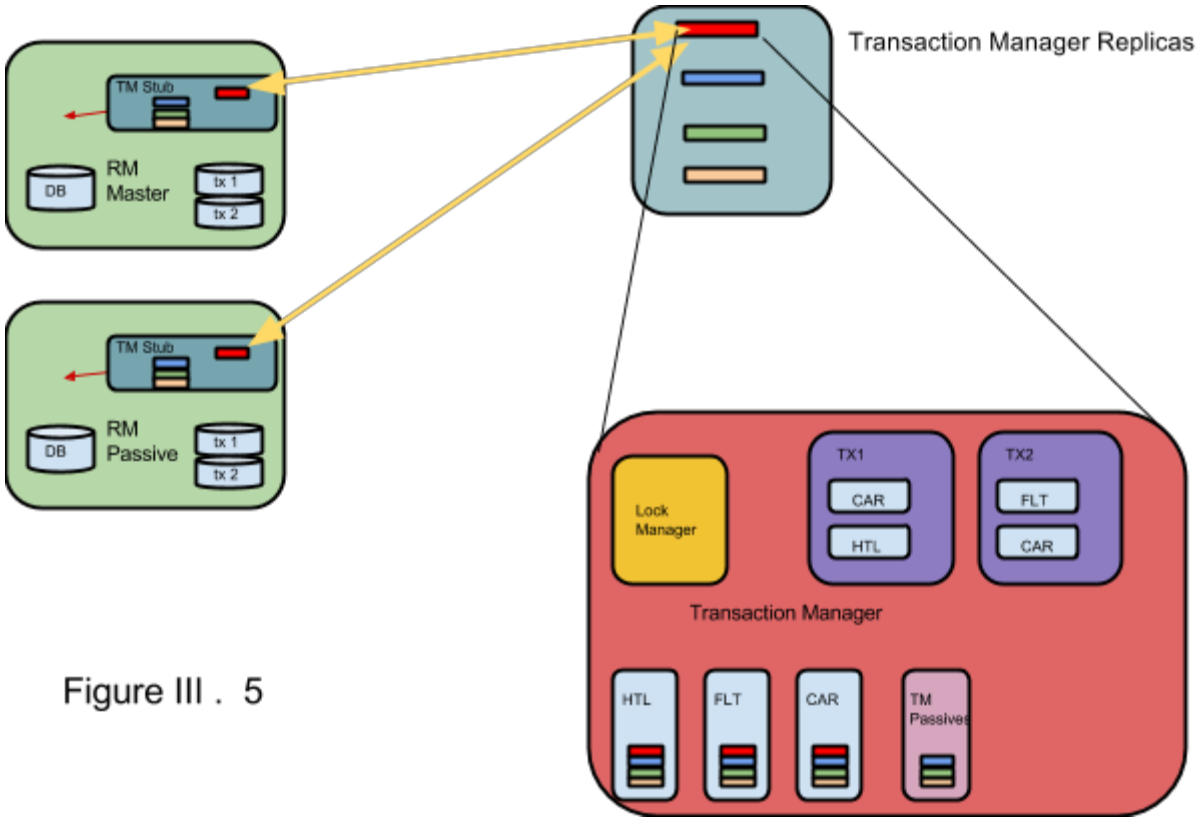


Figure III . 5

Transaction Manager Replication

A transaction manager on boot up as passive and looks up for a master and copies over the data from it. An exclusive lock is placed by the master when this data transfer happens. The master will from that point on, send any updates to it's data like lock, transaction state changes (commit/aborts), registration of new RM types and RMs to the replica via an application layer multicast mechanism. If no master or neighbours were found, it will declare itself as the master and start empty.

A master transaction manager always places a work lock before performing any non-replica startup requests. Multiple work locks can co-exist, where as a work lock and exclusive lock (used for transfer of master copy of data to a replica that's booting up) cannot co-exist.

Resource Manager Replication

A Resource manager on boot up as passive, registers with the TM as a replica for it's RM Type and looks up for a master and copies over the data from it. An exclusive lock is placed by the master when this data transfer happens. The master will from that point on, send any updates to it's data to the replica via an application layer multicast mechanism. If no master or neighbours were found, it will declare itself as the master and start with an empty database.

A master RM always places a work lock when it's doing any database updates. Multiple work locks can co-exist, however a work lock and an exclusive lock used for transferring data to another replica that's starting up cannot co-exist. This prevents any database level inconsistencies from occurring during data transfer.

An RM, on reception of a request, whether a master receiving request from Middleware, or a replica receiving the request from the Master, will check it has already processed it or not, the master will return the processed value if it finds it, the replicas will simply discard the request. Both master and replica stores the response to the request if it hasn't processed it before. Also a replica can look at the status send by master and decide to skip processing, if it sees that the master did not actually make any changes to its data.

Commits / Aborts are despatched by TM to the corresponding RM groups. An RM will update its database from txn buffers / discard based on the operation. This task is idempotent in nature. (i.e, multiple commits / aborts has no negative effect).

Crash interface

As an integrated approach to provide a common mechanism to mimic crash at various points, a CrashPoint interface was developed. A program can register valid crash points (string labels) with crash point interface when it starts up and then call the crash point interface at different locations in the program along with the label tagged to the location. The crash point interface will crash the program if that label is enabled for a crash.

A crash point client program is provided to connect to the program (via RMI) and view the available crash points as well as to turn on / off crash points.