Client:

```
    Config = getCurrentConfigFromOlympus();
    Create operationID
    Append < operationID, operation > to request
    sign(request)
    send("ClientRequest", request) to replica(Config.head);
```

Replica: on receiving ("ClientRequest" , request)

```
if (accept(shuttle)):

    Operation is < request.operationID, request.operation >
    if ( is Operation in seq<ResultShuttle> )
        generateResponseAfterReceivingResultShuttle(Operation, seq<ResultShuttle>,
History, LocalCache)
    else if (is replica not head)
        // forwarding request to head
        send("ClientRequest", request ) to replica(Config.head)
        if(await( is resultShuttle received for Operation ))
            generateResponseAfterReceivingResultShuttle(Operation,
        seq<ResultShuttle>, History, LocalCache)
        if( timeout(t))
            sendReconfigurationRequest(Operation)


    else if (is replica head)
        if ( is Operation in History.operation)
            if(await( is result shuttle received for Operation ))
                generateResponseAfterReceivingResultShuttle(Operation,
            seq<ResultShuttle>, History, LocalCache)
            else  if( timeout(t))
                sendReconfigurationRequest(Operation)

        else
            Create new shuttleObj
            shuttleObj.slot  is  History.lastSlotNumber + 1
            shuttleObj.operation is Operation
            actionsAtEveryReplica(shuttleObj,replica)
            send("ForwardShuttle", shuttleObj) to nextReplica
```

Replica: on receiving (“ForwardShuttle” , shuttle)

```
if (accept(shuttle)):
    actionsAtEveryReplica(shuttle,replica)
    if ( is replica not tail )
        send(“Shuttle”, shuttle) to nextReplica
    else
        result = getResultFromLocalCache()
        response = (result, shuttle.resultProof, shuttle.operation)
        sign(response)
        send(“BackwardShuttle”, shuttle ) to previousReplica
        send(“Response”, response ) to client
        Truncate shuttle
```

Replica: on receiving (“BackwardShuttle” , shuttle)

```
if (accept(shuttle)):
    result = getResultFromLocalCache()
    verifyResultAgainstResultProof(result,shuttle.resultProof,shuttle.operation)
    Cache shuttle into replica.seq<ResultShuttle>

    if ( is replica not head )
        send( “BackwardShuttle”, shuttle ) to previousReplica
    else
        if( shuttle.slot % N ==0)  // N is checkpointing frequency
            checkpointStatement.slot = shuttle.slot
            checkpointStatement.hashRunningState = getHashFromCheckpointRunningStateCache(shuttle.slot)
            sign(checkpointStatement)
            checkpointProof.append(checkpointStatement)
            send(“CheckPointProofForward”, checkpointProof) to nextReplica
        Truncate shuttle
```

Replica: on receiving (“CheckPointProofForward” , checkpointProof)

```
if (accept(checkpointProof)):
    checkpointStatement.slot = shuttle.slot
    checkpointStatement.hashRunningState = getHashFromCheckpointRunningStateCache(shuttle.slot)
    sign(checkpointStatement)
    checkpointProof.append(checkpointStatement)
    if(is replica not tail)
        send(“CheckPointProofForward”, checkpointProof) to nextReplica
    else
        send(“CheckPointProofBackward”, checkpointProof) to previousReplica
```

Replica: on receiving (“CheckPointProofBackward” , checkpointProof)

```
if (accept(checkpointProof)):
    lastCheckpointProof is checkpointProof
    Truncate History till checkpointProof.slot
    if(is replica not head)
        send(“CheckPointProofBackward”,checkpointProof) to previousReplica
    Else
        Truncate checkpointProof
```

Client : on receiving (“Response”, response)

```
if (verifySignature(response)):
    isValid =verifyResultAgainstResultProof(responce.result, response.resultProof, response.operation)
    if (isValid)
        Accept the result.
    else
        sendReconfigurationRequest(response.operation)
```

Replica: on receiving (“wedge” , wedgeRequest)

```
if (accept(wedgeRequest)):
    message.history = replica.History
    message.checkpointProof = replica.lastCheckpointProof
    message.seq<ResultShuttle> = replica.seq<ResultShuttle>
    send(“wedgedStatements”, message) to Olympus
    Change state to IMMUTABLE
```

Replica: on receiving (“catchUp” , operationToBePerformed)

```
if (accept(operationToBePerformed)):
    apply all operationToBePerformed to the current runningState.
    assign the cryptographicHash(runningState) to response
    send(“caughtUp”, response) to Olympus
```

Replica: on receiving (“getRunningState”, request)

```
if (accept(request)):
    assign current_running_state to response
    send(“getRunningStateResponse”,response) to Olympus
```

Replica: on receiving (“inithist”, configuration)

```
if (accept(configuration)):
    apply configuration.runningState to its runningState
    history = []
    Change state to ACTIVE
```

<u>Olympus: on receiving (“reconfiguration” , *reconfigurationRequest*)</u>

```
if (verifySignature(reconfigurationRequest)):
    create signed wedge request
    send(“wedge”, wedgeRequest) to all replicas

    seq wedgeStatements  contains all responses
    if(await( response from a quorum of replicas)):

        isValid = verifySignature() for each response in wedgeStatements
        if(isValid):
            // switchConfig:
            // ch is cryptographic hash of the accepted running state
            while(1):

                Quorum =
            selectQuorumWithValidHistoryAndCheckpoint(wedgeStatements))

                LH = getLongestSequenceOfOperations

                for each rho in Quorum
                    send(“catchUp”,(LH-wedgeStatements(rho).history))

                Wait for all caughtUp messages
                Let seq caughtUp contain all reponses

                if((ch = validateCaughtUpMessages(seq caughtUp ))!=-1)
                break

            while(1):
                send(“getRunningState”,dummySignedRequest) to next replica in
            this quorum
                Wait for “getRunningStateResponse”
                for runningState received:
                if(cryptographicHash(runningState) == ch)
                    C = new configuration with runningState assigned
                    assign  runningState to C
                    send(“inithist”, C) to all replicas
                    break
```

Method : generateResponseAfterReceivingResultShuttle(Operation, seq<ResultShuttle>, History, LocalCache)

```
    if(validateResultShuttleWithResultStoredInLocalCacheForTheOperation() == true)
        response.slot is History.slot associated with Operation
        response.result = getResultFromLocalCache(Operation)
        response.resultProof = getResultProofFromShuttle(Operation)
        response.operation = Operation
        sign(response)
        send("Response", response ) to client
    else
        sendReconfigurationRequest(shuttle.operation)
```

Method : actionsAtEveryReplica(shuttle, replica):

```
    isValid = Verify conflicting (shuttle.slot ,shuttle.operation) in replica.history &&
    && Verify that no holes exist for shuttle.slot
    && Verify if OrderStatements of all predecessors exist in shuttle.orderProof
    && Verify signs of shuttle.orderStatements

    if(isValid)
        result = apply shuttle.operation to replica.runningState
        store result in replica.LocalCache
        addOrderStatement(shuttle)
        crResult =  CryptographicHash(result)
        addResultStatement(shuttle, crResult)
        if( shuttle.slot % N ==0)  // N is checkpointing frequency.
            calculate hash of replica.runningState
            append <shuttle.slot,hash> in replica.checkpointRunningStateCache
        store (shuttle.slot , shuttle.operation, shuttle.orderProof ) in replica.History
    else
        sendReconfigurationRequest(shuttle.operation)
```

Method: sendReconfigurationRequest(operation):

```
    create  reconfigurationRequest
    assign operation to reconfigurationRequest
    sign(reconfigurationRequest)
    send("reconfiguration", reconfigurationRequest) to olympus
```

Method: addOrderStatement(shuttle)**:**

```
OrderStatement= ("order", shuttle.slot , shuttle.operation )
sign(OrderStatement)
Append OrderStatement to shuttle.seq<orderProof>
```

<u>**Method**: addResultStatement(shuttle, crResult):</u>

```
ResultStatement = ("result", shuttle.operation, crResult)
sign(ResultStatement)
Append ResultStatement to shuttle.resultProof
```

<u>**Method**: verifySignature(message):</u>

```
if  (message is signed by either client,olympus or replicas)
    return  true
else
    return false
```

<u>**Method**: accept(message, replicaState):</u>

```
if  (message is signed by either client or replicas) &&  replicaState == ACTIVE
    return  true
else if  message is signed olympus
    return true
else
    return  false
```

<u>**Method**:validateCaughtUpMessages(seq caughtUp):</u>

```
if( all the cryptographicHash(runningState) contained in seq caughtUp are same)
    return  cryptographicHash(runningState)
else
    return -1
```

<u>**Method**: selectQuorumWithValidHistoryAndCheckpoint(wedgeStatements):</u>

```
Find quorum 'Q' of t+1 replicas ,such that following two condition holds.
1  For each pair of replicas r1 and r2 in Q, for each slot 's' for which an order proof
appears in r1..history and r2.history, the order proofs for 's' are consistent, i.e., are for
the same operation
2  For each pair of replicas r1 and r2 in Q, checkpoint proofs associated with them are
consistent, i.e.,  crytographic hash of running_state across the proofs should be same.
return Q
```